VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

NOVÉ METODY ROZLOŽENÍ VE ZOBRAZOVACÍM STROJI CSS

NEW LAYOUT METHODS IN A CSS RENDERING ENGINE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. ONDŘEJ NOVÁK**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. RADEK BURGET, Ph.D.**
SUPERVISOR

BRNO 2021

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2020/2021

Zadání diplomové práce

Student: **Novák Ondřej, Bc.**
Program: Informační technologie
Obor: Informační systémy a databáze
Název: **Nové metody rozložení ve zobrazovacím stroji CSS**
New Layout Methods in a CSS Rendering Engine
Kategorie: Softwarové inženýrství
Zadání:

1. Seznamte se s projektem experimentálního zobrazovacího stroje CSSBox, jeho architekturou a způsobem výpočtu rozložení prvků.
2. Prostudujte nové způsoby rozložení obsahu pomocí jazyka CSS jako např. grid layout a flexbox a jejich existující implementace v jazyce Java.
3. Navrhněte způsob integrace nových způsobů rozložení obsahu do knihovny CSSBox.
4. Po dohodě s vedoucím proveďte úpravy existujících řešení a jejich integraci do projektu CSSBox.
5. Proveďte testování vytvořeného rozšíření pomocí vhodné testovací sady.
6. Zhodnoťte dosažené výsledky.

Literatura:

- Novák, Ondřej. Implementace mřížkového rozložení ve zobrazovacím stroji CSS. Brno, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií.
- Ondrák, Lukáš. Implementace flexibilního rozložení ve zobrazovacím stroji CSS. Brno, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií.
- Dokumentace projektu CSSBox: http://cssbox.sourceforge.net/documentation.php
- Michálek, M.: Vzhůru do CSS3, e-kniha, 2015, https://www.vzhurudolu.cz/ebook-css3

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 3

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/
Vedoucí práce: **Burget Radek, Ing., Ph.D.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2020
Datum odevzdání: 19. května 2021
Datum schválení: 22. října 2020

Abstrakt

Cílem této práce je prozkoumat experimentální zobrazovací stroj CSSBox, prostudovat nové způsoby rozložení obsahu webových stránek, kterými jsou mřížkové a flexibilní rozložení, za pomoci jazyka CSS a navrhnout příslušné úpravy pro jejich integraci do knihovny CSSBox. Nejdříve je shrnut jazyk CSS, včetně podrobných popisů nových rozložení, po kterých následuje návrh samotného řešení. Práce se dále věnuje podrobnému popisu implementační části, která je poté otestována na testovací sadě. Závěr práce zhodnocuje výsledky včetně nahlášených nedostatků a budoucích možnostech rozšíření.

Abstract

The goal of this thesis is to explore the CSSBox experimental rendering engine, to study new ways of layout the content of web pages, which are Grid layout and Flexbox layout, using CSS language and to design appropriate modifications for their integration into CSSBox library. At first, the CSS language is described, including detailed descriptions of the new layouts, followed by a proposal for the solution itself. The thesis also deals with a detailed description of the implementation part, which is then tested on a test set. The conclusion of the thesis evaluates the results, including the reported shortcomings and other possibilities for development.

Klíčová slova

Mřížkové rozložení, Flexibilní rozložení, CSSBox, CSS, Java

Keywords

Grid layout, Flexible layout, CSSBox, CSS, Java

Citace

NOVÁK, Ondřej. *Nové metody rozložení ve zobrazovacím stroji CSS*. Brno, 2021. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Radek Burget, Ph.D.

Nové metody rozložení ve zobrazovacím stroji CSS

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Radka Burgeta Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Ondřej Novák
18. května 2021

Poděkování

Tímto bych velmi rád poděkoval svému vedoucímu práce Ing. Radku Burgetovi, Ph.D. za cenné rady, veškerou pomoc a poskytnuté konzultace při zpracování této práce.

Obsah

Kapitola 1

Úvod

V současné době využívání webových prohlížečů již patří ke každodenní činnosti za účelem získávání informací, které jsou publikovány na webu. Je tedy nutné se věnovat vývoji nástrojů, které pomáhají k lepšímu zpracování informací a analýze jednotlivých webových stránek. Jedním ze zástupců těchto nástrojů je výzkumný projekt CSSBox.

CSSBox představuje experimentální zobrazovací stroj napsaný v jazyce Java a vznikl na Fakultě informačních technologií Vysokého učení technického v Brně. Tento stroj poskytuje informace o obsahu webové stránky a dále se využívá k testování algoritmů pro analýzu webových stránek.

Od vzniku tohoto projektu došlo k podstatnému vývoji v oblasti technologií běžně používaných pro implementaci webových stránek, především v oblasti kaskádových stylů CSS. Kromě jiných věcí přibyly zejména nové způsoby rozložení obsahu na webové stránce. Jedná se o nové typy moderního responzivního rozložení, kterými jsou mřížkové (Grid layout) a flexibilní (Flexible box layout) rozložení. Těmito novými rozloženími se již zabývaly dvě bakalářské práce [14] [15], jejichž hlavním cílem bylo experimentálně implementovat tyto rozložení v knihovně CSSBox.

Nicméně se však ukázalo, že původní architektura knihovny CSSBox není tak úplně vhodná pro takovéto rozšíření. Cílem této práce je proto navrhnout potřebné změny v architektuře knihovny CSSBox a implementovat tyto změny ve vnitřní reprezentaci dokumentů. Dalším cílem práce je rozšíření funkčnosti knihovny CSSBox o nové rozložení Grid layout a Flexbox. Toto rozšíření je dále otestováno na testovací sadě, kdy jsou poté zhodnoceny výsledky a nastíněny další možnosti rozšíření. Jedním z cílů je také provedení reorganizace kódu důležitých boxů kvůli oddělení vnitřní logiky boxu od zpracování jeho rozložení, které přinese lepší čitelnost kódu.

Kapitola 2

Kaskádové styly CSS

V této kapitole jsou nejprve popsány kaskádové styly a jejich syntaxe zápisu. Následuje krátké shrnutí jednotlivých úrovní kaskádových stylů. Poté se tato kapitola věnuje popisu CSS Box Modelu, ze kterého rovněž vychází i pozicování v knihovně CSSBox. Konec kapitoly se zabývá jednotlivými typy rozložení stránek, které se již nepoužívají, nebo jsou využívány dodnes.

2.1 O kaskádových stylech

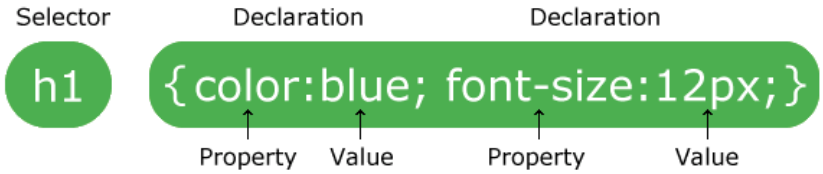
Kaskádové styly CSS[17] (angl. Cascading Style Sheets) popisují vzhled webových stránek vytvořených pomocí značkovacích jazyků, kterými jsou například HTML či XML. CSS jsou považovány za deklarativní jazyk, který vytvořila společnost W3C (World Web Consortium) v roce 1996. Přidávají různé vlastnosti týkající se elementů, jako jsou například barvy, velikosti, stylování textu, animace, pozadí nebo různé rozvržení stránky. Kaskádové styly lze zapsat přímo do kódu jazyka HTML (vložené styly), konkrétněji jako atribut *style* daného elementu. Další možností je zapsání do záhlaví stránky (interní styly) pomocí značek *<style>...</style>*. Poslední a nejvíce používanou možností je psaní stylů do externích souborů (externí styly) s příponou **.css*, na které se pak stránka odkazuje pomocí značky *<link>*. Tyto externí soubory je možné pak libovolně využívat pro více stránek současně.

Syntaxe kaskádových stylů

Syntaxe[19] kaskádových stylů je reprezentována pomocí tzv. pravidel (angl. rules), které jsou složeny ze selektoru a skupiny deklarací. Příklad takového pravidla je znázorněn na obrázku 2.1. Každá skupina deklarací je obalena ve složených závorkách *{...}* a obsahuje jednotlivé deklarace (oddělené středníkem) složené z vlastnosti CSS a její příslušné hodnoty (oddělené dvojtečkou). Selektory se používají pro výběr prvků jazyka HTML, které budou stylovány, a díky nim se ví, jaké styly mají jednotlivé prvky. V kaskádových stylech existuje několik typů selektorů, mezi které lze zařadit jednoduché (základní) selektory, pseudotřídy, pseudoelementy a atributové selektory. Jednoduché selektory jsou následující:

- Element selektor – vybíhá jednotlivé elementy jazyka HTML na základě jejich názvu. Příkladem může být *p, td, h2*.
- ID selektor – využívá se k výběru elementu s atributem *id*, hodnota tohoto atributu na stránce by měla být jedinečná. ID selektor se zapisuje pomocí znaku # a hodnotou atributu *id*. Příkladem může být *#foot*.

- Class selektor – využívá se k výběru elementů s atributem *class*, hodnota tohoto atributu již nemusí být jedinečná. Class selektor se zapisuje pomocí znaku . a hodnotou atributu *class*. Příkladem může být *.container*.



Obrázek 2.1: Syntaxe používající se v kaskádových stylech. Jedná se o pravidlo složené ze selektoru a dvou deklarací. Obrázek byl převzat ze zdroje [19].

2.1.1 Úrovně kaskádových stylů

Následující podkapitola lehce shrne jednotlivé úrovně[10][18] kaskádových stylů. Nastíní také, jaké nové vlastnosti s sebou přinesly. Důležité také je, že vlastnosti starších úrovní nemusí být podporovány v novějších úrovních, nebo jsou již zrušeny.

CSS Level 1

Jak již bylo řečeno, kaskádové styly byly vytvořeny v roce 1996, vznikla tedy první úroveň těchto stylů. Tato úroveň obsahovala základní vlastnosti pro písmo, barvy a pozadí. Zároveň byly představeny vlastnosti pro mezery mezi slovy nebo mezery mezi řádky. Bohužel webové prohlížeče tuto úroveň dlouhou dobu nepodporovaly.

CSS Level 2

Tato úroveň byla publikována v roce 1998 a jedná se o nadmnožinu úrovně CSS 1. Mezi významné možnosti této úrovně patří vlastnosti pro rozložení stránky, jako je relativní, pevné a absolutní umístění jednotlivých elementů. Byly představeny styly pro zvukovou podporu a další typy selektorů. Rovněž byl představen koncept typů médií, což přineslo možnost použít různé styly pro různé typy médií. Byla vydána i úroveň CSS 2.1, která opravuje chyby nalezené v úrovni CSS 2.

CSS Level 3

Mezi hlavní funkce této úrovně patří například vlastnosti prezentačního charakteru, což umožňuje efektivní vytváření prezentací z webových dokumentů. Do těchto vlastností může patřit: kulaté rohy, vlastní písmo, stínové efekty, apod. Tato úroveň rovněž přinesla podporu animací a jednoduchých transformací. Byly přidány i nové selektory kvůli efektivnějšímu stylování a tzv. dotazy na média. Úroveň CSS 3 je rozdělena na několik samostatných dokumentů, které se označují jako moduly. Díky tomuto rozdělení má každý modul svůj stav a stabilitu.

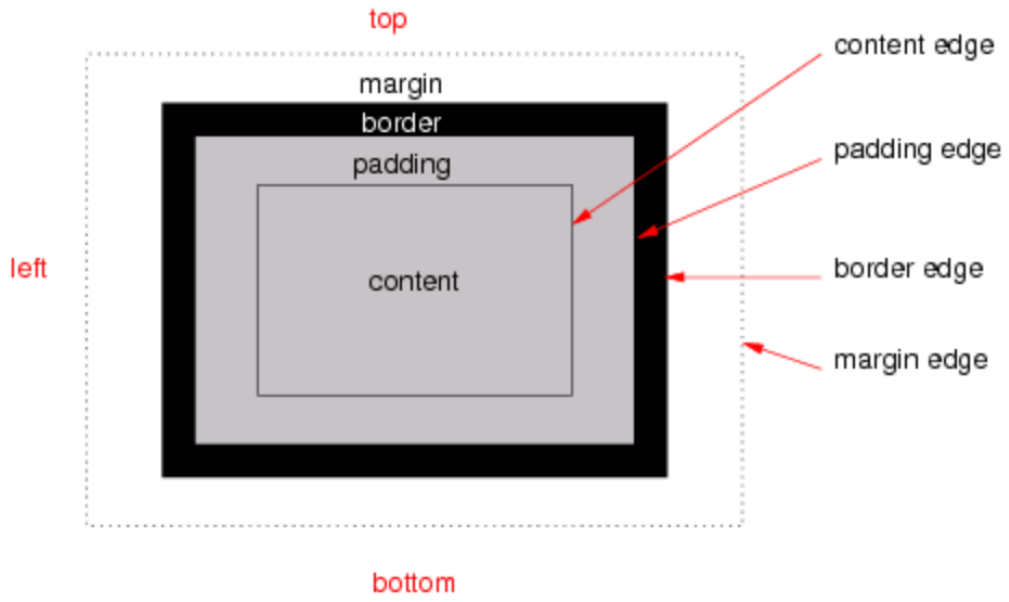
CSS Level 4

Co se týče této úrovně, tak zatím neexistuje žádný standard, který by byl pojmenován jako CSS 4. Úroveň CSS 4 s největší pravděpodobností ani nebude, protože již úroveň CSS

3 přišla s rozdělením do jednotlivých modulů a ty jsou vyvíjeny nezávisle na sobě. Již existují některé moduly s úrovní 4, které jsou postaveny na předchozích modulech úrovně 3. Souhrnně lze tyto moduly s úrovní 4 označovat jako CSS Level 4, ale jedná se pouze o označení.

2.2 CSS Box Model

CSS Box Model[9] představuje základ jednotlivých elementů na webových stránkách. Každý element je reprezentován pomocí obdélníkové oblasti. Ta má svůj obsah (angl. content) tvořený výškou a šířkou, který obsahuje například text, další boxy, obrázek, apod. Dále obsahuje další volitelné vlastnosti, kterými jsou *padding*, *border* a *margin*. Těmto vlastnostem lze určit jejich velikost, ta je v základním nastavení nulová. U vlastnosti *margin* je možné její hodnotu nastavit i na záporné hodnoty. Obrázek 2.2 znázorňuje CSS Box model s jeho vlastnostmi.



Obrázek 2.2: CSS Box model s vlastnostmi, který se využívá pro jednotlivé elementy. Obrázek byl převzat ze zdroje [9].

Vlastnosti *padding*, *border* a *margin* lze dále rozdělit na horní (angl. top), spodní (angl. bottom), levý (angl. left) a pravý (angl. right) segment. Každý z těchto segmentů je možné ovládat pomocí odpovídajících vlastností CSS a má i svůj obvod tvořený čtyřmi hranami (angl. edges).

2.3 Typy rozložení

Rozložení[16] (angl. layout) představuje vzor, pomocí kterého se definuje struktura webové stránky. Rozložení se dělí na dva typy, kde první typ představuje dělení podle počtu sloupců, a druhý typ představuje dělení podle nastavení šířky stránky. Nyní následuje popis vybraných typů rozložení.

Tabulkové rozložení

Tabulkové rozložení[6] (angl. table layout) využívá tabulky pro pozicování. Jedná se o základní rozložení, na které stačí obyčejný jazyk HTML, kdy ještě neměly prohlížeče podporu CSS. Toto rozložení je například jednoduché a kompatibilní. V opačném případě je toto rozložení poměrně pomalé, protože je tabulka zobrazena až po úplném načtení, je zde i neúspornost kódu a špatná přenositelnost. Tabulkové rozložení se již dnes skoro nepoužívá, protože jsou spíše využívány rozložení s velkou podporou CSS.

Blokové rozložení

Blokové rozložení[6] (angl. block layout) využívá pro pozicování dvou elementů, kterými jsou element * * a **. Jedná se o elementy, které se dají formátovat pomocí jakékoliv vlastnosti CSS. Element * * se převážně využívá pro rozložení výsledné stránky a element ** zase pro formátování textu. Výsledný kód je mnohem přehlednější a rychlejší než v případě tabulkového rozložení, tudíž se i rychleji načítá. Toto rozložení bylo populární a využívá se až dodnes.

Fixní a fluidní rozložení

Fixní rozložení (angl. fixed layout) používá předem nastavenou velikost šířky, která je fixní a nemění se vůči prohlížeči. Předem nastavená velikost šířky je nejčastěji zadána v pevných jednotkách (například pixely). U tohoto typu rozložení se často objevuje nežádoucí horizontální rolovací lišta a již se tento typ nepoužívá.

Fluidní rozložení (angl. fluid layout) na rozdíl od fixního využívá již relativní jednotky nebo procenta. Oproti fixnímu rozložení pracuje se šířkou prohlížeče a mění se podle ní. Toto rozložení má již lepší vlastnosti, ale stále má své problémy ohledně malých nebo velkých oken (výsledek nemusí vypadat pěkně).

Responzivní rozložení

Responzivní rozložení (angl. responsive layout) používá dotazy na média a relativní jednotky. Snaží se přizpůsobit šířce okna a musí zaručit, že obsah bude bezchybně zobrazen na různých zařízeních. Při zmenšování či zvětšování okna se obsah vždy přizpůsobí. Pokud by byla překročena určitá hranice šířky definována pomocí dotazů na média, výsledný obsah bude více změněn tak, aby nejlépe odpovídal danému oknu. Tento typ rozložení se v posledních letech stal velmi populární a do tohoto typu patří například mřížkové nebo flexibilní rozložení.

Kapitola 3

Grid layout

CSS Grid layout[8][1] (mřížkové rozložení), zkráceně grid, představuje nový modul CSS a je zařazen jako další typ responzivního rozložení webové stránky. Jedná se o dvojrozměrný systém rozložení obsahu založený na mřížce. Mřížka je tvořena za pomoci linek, které rozdělují celou mřížku na určité oblasti. Do těchto oblastí lze umisťovat jednotlivé položky mřížky.

Cílem Grid layoutu je úplně změnit způsob, jakým se navrhují uživatelská rozhraní, která jsou založena na mřížce. Nejprve se totiž využívalo způsobu na bázi tabulek, později se přišlo s tzv. floaty a následně bylo přidáno ještě pozicování a *inline-bloky*. Nicméně všechny tyto metody vynechaly několik důležitých funkcí (příkladem může být vertikální centrování). Jako další metodou byl vyvinut Flexible box layout (viz. Kapitola 4), který měl tyto nedostatky řešit, ale nakonec byl určen pro jednodušší jednorozměrné rozložení. Grid layout je tedy první modul, speciálně vytvořený, pro řešení problémů s rozložením, které předchozí metody vynechaly. Grid layout se velmi často využívá s Flexible box layoutem, protože spolu tyto moduly velmi dobře spolupracují.

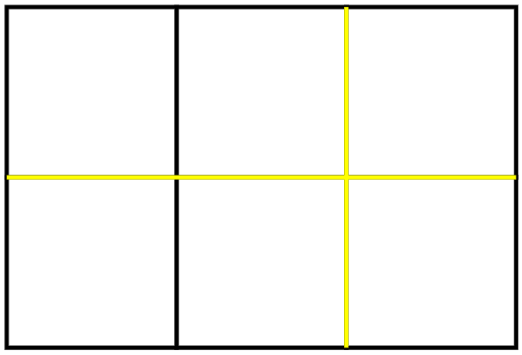
3.1 Základní terminologie

Grid layout zavádí několik nových termínů, které je potřeba znát a porozumět jim. Tato kapitola se těmito termíny zabývá, vyskytují se zde pojmy jako linka mřížky, stopa mřížky, buňka mřížky, oblast mřížky a mezery mřížky. U každého vysvětleného termínu je i navíc přidán obrázek pro lepší ilustraci a představu. V poslední části této kapitoly jsou rozebírány nové jednotky velikosti, které jsou ve spojení s Grid layoutem hojně využívány.

Linka mřížky

Linka mřížky[2] (angl. grid line) je základem pro modelování mřížky, protože se pomocí ní vytváří celá mřížka. V mřížce se využívá dvěma způsoby, buď jako vertikální směr, nebo horizontální směr. Znázorněné linky žlutou barvou v mřížce lze vidět na obrázku 3.1.

Každá linka v mřížce je dostupná pomocí číselného indexu, který se začíná indexovat od čísla jedna. Místo číselného indexu je možné využít i vlastní pojmenování linek. Nejčastěji se tento způsob používá pro lepší přehlednost, kdy se lince specifikuje její význam v mřížce. Každá položka mřížky (viz. Kapitola 3.3) je pak pomocí odkazů na konkrétní linky umístěna na specifické místo v mřížce.

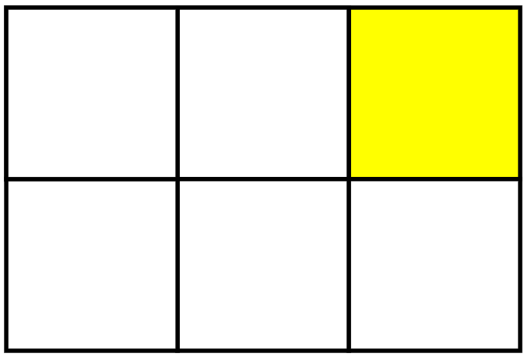


Obrázek 3.1: Horizontální linka s indexem 2 a vertikální linka indexována číslem 3. Obrázek byl upraven a převzat ze zdroje [1].

Buňka mřížky

Pojmem buňka mřížky[2] (angl. grid cell) se rozumí prostor mezi čtyřmi linkami uvnitř mřížky. Obrázek 3.2 vyobrazuje právě takovou buňku, ta je znázorněna žlutou barvou.

Buňka představuje nejmenší jednotku celé mřížky, na kterou se lze odkazovat, jestliže umisťujeme do mřížky jednotlivé položky. Konceptuálně se dá považovat za buňku tabulky, která se typicky tvoří pomocí značky `<td>`.

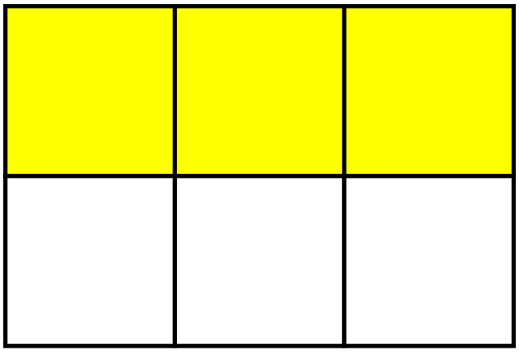


Obrázek 3.2: Buňka mřížky vyskytující se mezi horizontálními linkami s indexy 1 a 2 a mezi vertikálními linkami s indexy 3 a 4. Obrázek byl upraven a převzat ze zdroje [1].

Stopa mřížky

Pojem stopa mřížky[2] (angl. grid track) představuje obecný termín pro řádek respektive sloupec mřížky. Jinými slovy tento pojem lze vyjádřit jako prostor mezi sousedními linkami v horizontálním či vertikálním směru. Příkladem takové stopy mřížky (znázorněna žlutou barvou) může být obrázek 3.3.

Nicméně každá stopa má přiřazenou funkci dimenzování udávající, jak široký má být sloupec nebo jak vysoký má být řádek. Jinak řečeno, jak mají být od sebe vzdáleny linky mřížky. V poslední řadě je dobré zmínit, že sousední stopy lze od sebe oddělit tzv. mezerami (viz. dále na straně 10).

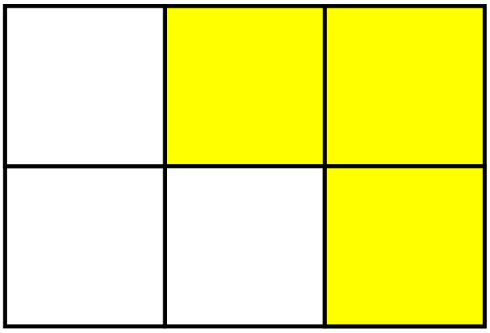


Obrázek 3.3: Znázorněná horizontální stopa mřížky uvnitř linek s indexem 1 a 2. Obrázek byl upraven a převzat ze zdroje [1].

Oblast mřížky

Oblast mřížky[2] (angl. grid area) slouží jako logický prostor, který se využívá pro rozložení jedné nebo více položek mřížky. Může se skládat z jedné nebo i více sousedních buněk mřížky. Jedná se o vazbu několika linek, kde jedna linka se vyskytuje na každé straně oblasti, a zároveň se podílí na dimenzování těch stop mřížky, které protíná. Vzorová oblast mřížky (znázorněna žlutou barvou) je představena na obrázku 3.4.

Oblasti mřížky se často využívají při tvorbě vzhledu stránky. Jelikož se dají jednoduše explicitně pojmenovat, lze pomocí nich snadno navrhnout výsledný vzhled stránky.



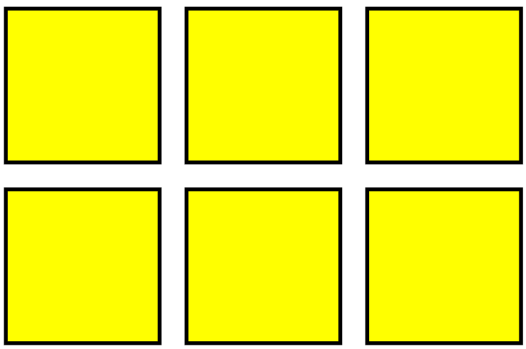
Obrázek 3.4: Oblast mřížky zabírající převážně pravou horní část celé mřížky. Obrázek byl upraven a převzat ze zdroje [1].

Mezery mřížky

Mezery mřížky[2] (angl. grid gutters) nebo také tzv. žlaby mřížky slouží k definování mezer (pokud jsou explicitně zadány) mezi sloupci a řádky v mřížce. Jakmile nejsou nastaveny, neobsahují žádnou hodnotu a chovají se jako mezery s nulovou velikostí. Efekt těchto mezer se dá vysvětlit tak, že všechny ovlivněné linky mřížky získají určitou tloušťku. Názorná ukázka mezer mřížky je představena na obrázku 3.5.

Kvůli dimenzování je každá mezera mřížky považována jako extra prázdná stopa s předem zadanou velikostí. Nicméně tyto mezery se vyskytují pouze uvnitř implicitní mřížky, tedy před první a za poslední stopou již žádné mezery nejsou. Při umisťování položek do

mřížky přes více než jednu stopu je potřeba zohlednit všechny nenulové mezery a připočítat je k celkové velikosti dané položky.



Obrázek 3.5: Mezery mezi položkami v celé mřížce.

Nová jednotka velikosti 'fr'

Grid layout s sebou nese i novou jednotku pro velikost sloupců a řádků.[13] Jedná se o jednotku *fr*, tato zkratka nejspíše vznikla z angl. slova fraction (zlomek), jelikož představuje zlomek zbývajícího místa uvnitř kontejneru. Tuto jednotku můžeme spočítat jako poměr jedné části k poměru součtu všech částí. Lze si tento výpočet představit pomocí vzorce 3.1.

Tato jednotka také zhruba odpovídá hodnotám, které se ve Flexboxu využívají u vlastností *flex-grow* a *flex-shrink*.

$$FR\_unit = \frac{available\_space - tracks\_sum}{fraction\_sum} \tag{3.1}$$

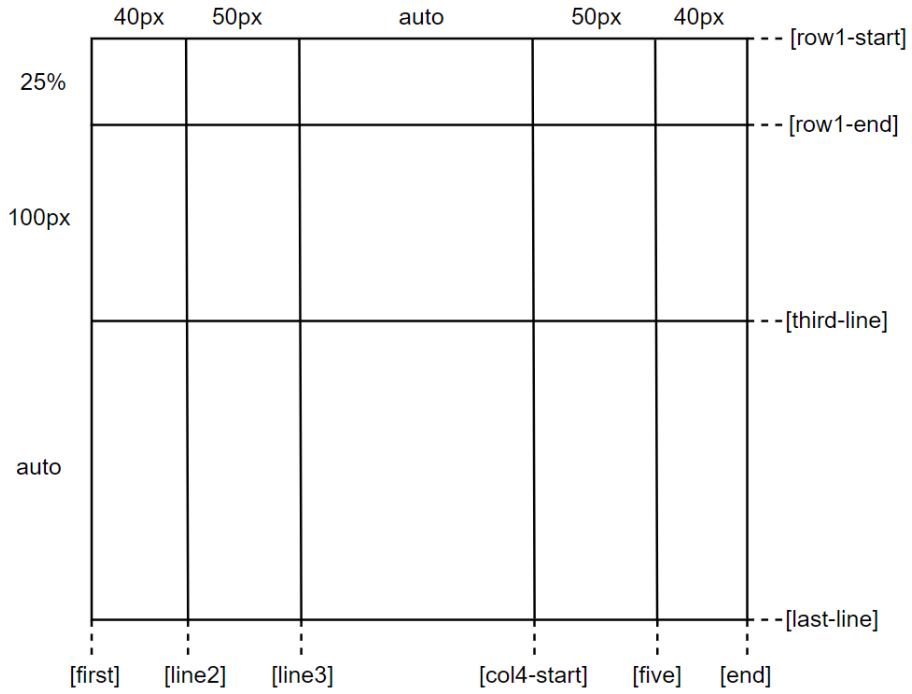
- FR_unit – vypočtená velikost jednotky fr
- available_space – dostupný prostor, kde by se mohla jednotka fr roztáhnout
- tracks_sum – součet všech stop mřížky, které mají pevnou velikost
- fraction_sum – součet všech jednotek fr ve stopách

3.2 Kontejner mřížky

Na začátku je důležité zmínit, že kontejner mřížky[2] (angl. grid container) s sebou přináší dvě nové hodnoty vlastnosti *display* v kaskádových stylech. První z těchto hodnot představuje `display: grid`, ta způsobuje, že element vygeneruje kontejner mřížky, který se v rozložení stránky chová, jako by se jednalo o blokový element. Druhá ze zmíněných hodnot je `diplay: inline-grid`, ta rovněž způsobuje, že je vygenerován kontejner mřížky, ten se ale chová v rozložení stránky jako inline element.

Kontejner mřížky pro svůj obsah vytváří nový nezávislý kontext formátování mřížky. Jako vysvětlení se dá považovat, že se jedná o vytváření kontextu blokového formátování s tím rozdílem, že se místo blokového rozložení využije mřížkové rozložení, tedy floaty se

nedostávají do kontejneru mřížky a vlastnost *margin* se nespojuje s vlastnostmi (rovněž se jedná o vlastnost *margin*) položek mřížky. Jak již bylo řečeno, tak obsah kontejneru mřížky je rozložen na mřížku, kde jednotlivé linky vytvářejí hranice pro každou položku mřížky, která se nachází uvnitř této mřížky. Tyto hranice také ale platí i pro vnořené bloky, které se nacházejí uvnitř dané položky. Pro názorný příklad je kontejner mřížky vyobrazen na obrázku 3.6.



Obrázek 3.6: Znázorněný kontejner mřížky, který mimo definovaných velikostí stop obsahuje i pojmenované jednotlivé linky. Obrázek byl upraven a převzat ze zdroje [8].

Jelikož není kontejner mřížky považován za blokový kontejner (nejsou si rovni) tak některé vlastnosti, které jsou navrženy pro blokové rozložení, nemůžou být aplikovány na mřížkové rozložení. Jedná se právě o tyto vlastnosti:

1. `float` – tato vlastnost nemá žádný vliv na položky mřížky, ale stále ovlivňuje hodnotu `display` u vnořených kontejnerů mřížky, protože se tato hodnota vypočítává před určením položek v mřížce.
2. `clear` – tato vlastnost nemá vůbec žádný vliv na jakékoliv položky v mřížce.
3. `vertical-align` – tato vlastnost rovněž nemá žádný vliv na položky uvnitř mřížky.
4. pseudo-elementy `::first-line` a `::first-letter` vůbec neovlivňují kontejner mřížky a nijak nepřispívají svým předkům.

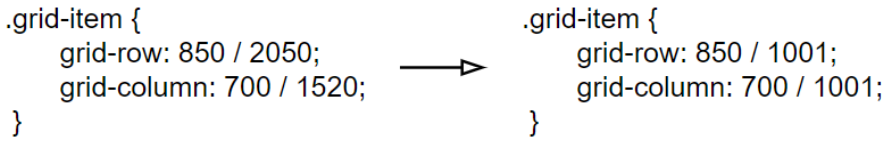
Za zmínku také stojí fakt, že pokud hodnota *display* kontejneru mřížky je nastavena na `display: inline-grid` a zároveň je tento kontejner plovoucí nebo absolutně umístěný, tak se hodnota vlastnosti *display* automaticky přepočítává na `display: grid`.

Omezení velkých mřížek kontejneru

Jelikož není paměť nekonečná, ale stále je omezená, můžou UA[1] upnout možnou velikost implicitní mřížky tak, aby byla rozložena v definovaném limitu UA (ten by měl být schopen pojmout linky v rozmezí od -10 000 do 10 000). Pokud by se ale nějaké linky nacházely mimo tento definovaný limit, budou zrušeny. Toto omezení kontejneru se týká především položek, kde mohou nastat dva případy:

1. Je-li položka rozprostřena mimo omezenou velikost mřížky, je její velikost upnuta na poslední možnou linku mřížky. Tento případ je znázorněn na obrázku 3.7.

2. Je-li položka rozprostřena zcela mimo omezenou velikost mřížky, je její velikost zmenšena na jedna a je přemístěna do poslední stopy v mřížce.



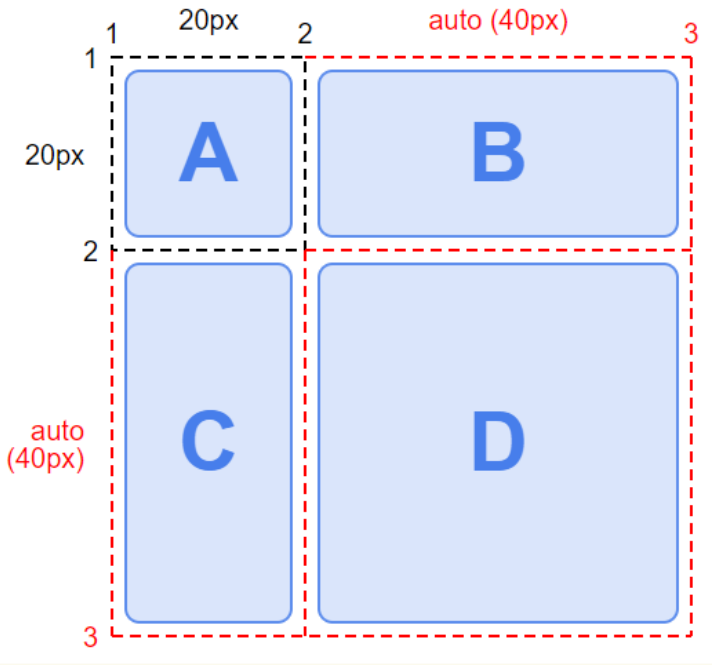
Obrázek 3.7: Pokud UA má podporu mřížek s maximálním počtem 1000 stop v každé dimenzi a kus položky je mimo tento prostor, budou automaticky koncové linky položky přepočítány na hodnotu 1001.

3.2.1 Implicitní mřížka

Úvodem je potřeba říci, že implicitní mřížka[2] existuje po celou dobu využívání Grid layoutu. Přesněji tato věta značí to, že jakmile se danému kontejneru mřížky specifikuje vlastnost CSS `diplay: grid`, implicitní mřížka je ihned vytvořena.

Jakmile je nějaká položka mřížky umístěna za hranice explicitní mřížky, kontejner ihned automaticky generuje další potřebné linky mřížky. Tyto linky (díky linkám se vytvářejí automaticky i stopy mřížky) tvoří dohromady s explicitní mřížkou právě implicitní mřížku. Obrázek 3.8 znázorňuje implicitní mřížku, která vznikla spojením explicitní mřížky a implicitně generovanými stopami.

[1] UA – user agent



Obrázek 3.8: Mřížka o velikosti 2x2 obsahující jednu explicitní buňku mřížky s rozměry 20x20 pixelů a další tři buňky vytvořené implicitně pro nové položky mřížky. Obrázek je převzat ze zdroje [2].

Mezi vlastnosti CSS, které se spojují s implicitní mřížkou, patří: `grid-auto-columns`, `grid-auto-rows`, `grid-auto-flow`, `grid` a také `gap`. Následující odstavce se budou zabývat popisem právě těchto vlastností.

Vlastnosti grid-auto-rows a grid-auto-columns

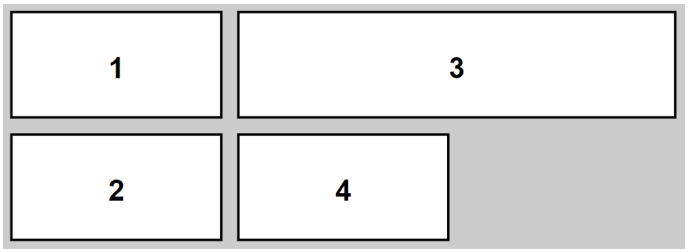
Vlastnosti `grid-auto-rows` a `grid-auto-columns` definují velikosti stop v řádcích či sloupcích. Vytvářejí tedy nové řádky či sloupce v mřížce pro položky mřížky, pokud jsou potřeba (položky se již nevejdou do explicitní mřížky). Tyto vlastnosti se dále využívají při automatickém umístění položek, pokud se již položka nevejde na daný řádek či sloupec, jsou pro ni vytvořeny nové. V CSS se tyto vlastnosti zapisují následovně:

```
grid-auto-rows: 100px max-content auto;
grid-auto-columns: min-content 50px 30%;
```

Vlastnost grid-auto-flow

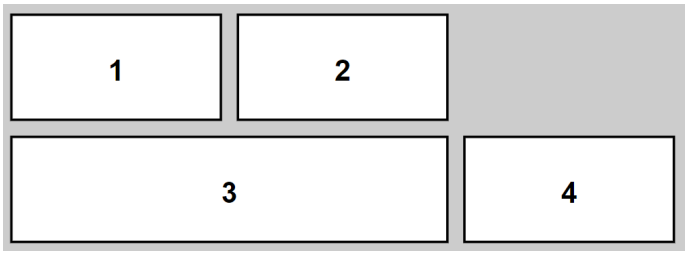
Vlastnost `grid-auto-flow` je využívána k automatickému umisťování jednotlivých položek mřížky, u kterých nebylo explicitně nastaveno jejich umístění. Položky jsou umisťovány do volného prostoru v mřížce pomocí následujících základních hodnot.

Hodnota *column* postupně umisťuje položky do volných prostorů ve sloupcích mřížky. Pokud již nezbývá žádný další volný prostor, automaticky je přidaný nový sloupec. Princip tohoto umisťování je znázorněn na obrázku 3.9.



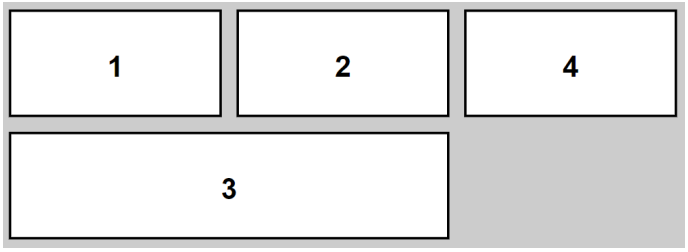
Obrázek 3.9: Umístěné položky za pomoci vlastnosti CSS `grid-auto-flow: column`.

Hodnota *row* postupně umisťuje položky do volných prostorů v řádcích mřížky. Pokud už není žádný volný prostor, je automaticky přidán nový řádek. Princip tohoto umisťování je znázorněn na obrázku 3.10.



Obrázek 3.10: Umístěné položky za pomoci vlastnosti CSS `grid-auto-flow: row`.

Hodnota *dense* představuje zajímavější způsob umisťování položek v mřížce. Pokouší se zaplňovat volné prostory dříve v mřížce, pokud by se později objevily menší položky, které by se do těchto prostorů vešly. Princip tohoto umisťování je znázorněn na obrázku 3.11.



Obrázek 3.11: Umístěné položky za pomoci vlastnosti CSS `grid-auto-flow: dense`.

Vlastnost grid

Vlastnost `grid` představuje zkratku, která dokáže nastavit všechny explicitní a implicitní vlastnosti mřížky v jedné deklaraci, ale nepodporuje vlastnost `gap`. Využívá se hlavně k vynechání dlouhých zápisů a také kvůli přehlednosti. V CSS se tato vlastnost dá zapsat například následovně:

```
grid: auto-flow 1fr / 100px;
```

Vlastnost gap

Vlastnost `gap`[2] definuje mezery mezi řádky a sloupci. Lze ji využít pouze v implicitní mřížce nebo ve spojení s explicitní mřížkou. Zároveň se také jedná o zkratku, která pouze zastupuje vlastnosti `row-gap` a `column-gap`. V CSS se tato vlastnost zapisuje následovně:

```
gap: 50px 100px;
```

3.2.2 Explicitní mřížka

Explicitní mřížka[2] se hojně využívá ve spojení s vytvářením vlastních layoutů webových stránek, kde si uživatel podle svých představ rozloží, jak by měla stránka vypadat. Způsob spočívá v tom, že se vytvářejí pojmenované oblasti, včetně jejich velikostí, a do těchto oblastí se následně ukládají jednotlivé položky. Explicitní mřížku lze v CSS definovat pomocí těchto čtyř vlastností: `grid-template-rows`, `grid-template-columns`, `grid-template-areas` a `grid-template`, tyto vlastnosti budou následně popsány.

Vlastnosti grid-template-rows a grid-template-columns

Vlastnosti `grid-template-rows` a `grid-template-columns` definují, jak velké budou řádky a sloupce explicitní mřížky. Dále slouží k určení, jak velká bude mřížka. Často se ve spojení s těmito vlastnosti využívají funkce `repeat()` a `minmax()`, aby se předešlo zdlouhavým zápisům. V CSS se tyto vlastnosti dají zapsat následovně:

```
grid-template-rows: 200px 2fr 30% minmax(30px, 100px);
grid-template-columns: repeat(4, minmax(50px, 1fr));
```

Vlastnost grid-template-areas

Vlastnost `grid-template-areas` slouží k vytváření pojmenovaných oblastí, na které se pak jednotlivé položky odkazují. Poskytuje také přehled o struktuře mřížky, což usnadňuje lepší pochopení rozložení kontejneru. Každý řádek této vlastnosti (stopa mřížky) je obalen ve speciálních znacích `""` kvůli tomu, aby bylo možné vytvořit více řádků. Pokud by vše bylo vepsáno pouze uvnitř dvou znaků `""`, jednalo by se pouze o jeden velký řádek. V CSS se tato vlastnost může například zapsat následovně:

```
grid-template-areas: "menu header header"
                     "menu content content"
                     "footer footer footer";
```

Vlastnost grid-template

Vlastnost `grid-template` představuje zkratku, shrnuje všechny předchozí vlastnosti do jedné deklarace. Využívá se kvůli jednoduchosti, kde si lze pomocí jedné vlastnosti nadefinovat celou mřížku. V CSS se tato vlastnost může zapsat následovně:

```
grid-template: "a a a" 40px
               "b c c" 40px
               "b c c" 40px / 1fr 1fr 1fr;
```

[2] dříve tato vlastnost byla pojemnována jako `grid-gap`, ale označení `grid-gap` se stále může používat

3.3 Položka mřížky

Položka mřížky[2] (angl. grid item) je považována za box, který představuje tzv. *in-flow*[3] obsah svého kontejneru. Jinými slovy je položka mřížky reprezentována jako potomek kontejneru. Položka mřížky samozřejmě může obsahovat i další vnořené elementy (například blokové a inline). Velikost těchto elementů je limitována velikostí dané položky. Nicméně pokud položka mřížky obsahuje kombinaci blokových a inline elementů, jsou inline elementy zabaleny do tzv. anonymních boxů[4]. To samé platí pro případ, pokud by položkou mřížky byl pouze samostatný text, tak tato položka je také zabalena to anonymního boxu, ale stále se s tímto boxem pracuje jako s položkou mřížky. Pokud by položka obsahovala pouze bílé znaky, tak její obsah nebude zobrazen, ale položka jako taková bude vykreslena.

3.3.1 Umístění položek do mřížky

Každá položka je spjata se svou oblastí nebo buňkou v mřížce. Jakmile je položka umístěna do oblasti nebo buňky, stane se z ní blok určující její polohu. Poloha položky je definována tedy svým umístěním, které se skládá ze souřadnic (automaticky nebo pevně zadané) a rozpětí (kolik stop položka zabírá v mřížce, výchozí hodnota je jedna).

Mezi vlastnosti CSS, které jsou spojovány s umisťováním položek do mřížky, patří: `grid-column-start`, `grid-column-end`, `grid-row-start`, `grid-row-end`, `grid-column`, `grid-row` a `grid-area`. Následující odstavce se budou zabývat krátkým popisem těchto vlastností.

Vlastnosti grid-column-start a grid-column-end

Vlastnosti `grid-column-start` a `grid-column-end` určují velikost a umístění položky ve sloupcích mřížky. Lze jim zadat pevnou velikost (případně automatickou) pro umístění nebo rozpětí přes kolik sloupců má být položka umístěna. V CSS se tyto vlastnosti dají využít například následovně:

```
grid-column-start: 2;
grid-column-end: span 3;
```

Vlastnosti grid-row-start a grid-row-end

Vlastnosti `grid-row-start` a `grid-row-end` určují velikost a umístění položky v řádcích mřížky. Lze jim zadat pevnou velikost (případně automatickou) pro umístění nebo rozpětí přes kolik řádků má být položka umístěna. V CSS se tyto vlastnosti dají využít například následovně:

```
grid-row-start: 2;
grid-row-end: -1;
```

[3] https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Flow_Layout/In_Flow_and_Out_of_Flow

[4] anonymní (annonymous) box — jedná se o box, který nelze řešit pomocí selektorů CSS a všechny jeho vlastnosti obsahují výchozí hodnoty

Vlastnosti grid-column a grid-row

Vlastnosti `grid-column` a `grid-row` zastupují zkratky předešlých vlastností. Vlastnosti pro počátek položky se píšou před znak lomeno a vlastnosti pro konec položky se píšou za znak lomeno. V CSS se tyto vlastnosti dají využít například následovně:

```
grid-column: span 2 / 7;
grid-row: auto / auto;
```

Vlastnost grid-area

Vlastnost `grid-area` rovněž představuje zkratku. Jedná se o zkratku zastupující všechny předešlé vlastnosti, které lze shrnout pouze na jeden řádek zápisu. Zároveň pomocí všech těchto vlastností je možné položky umisťovat i do pojmenovaných oblastí. V CSS se tato vlastnost dá využít například následovně:

```
grid-area: 3 / 3 / auto / span 4;
```

Kapitola 4

Flexible box layout

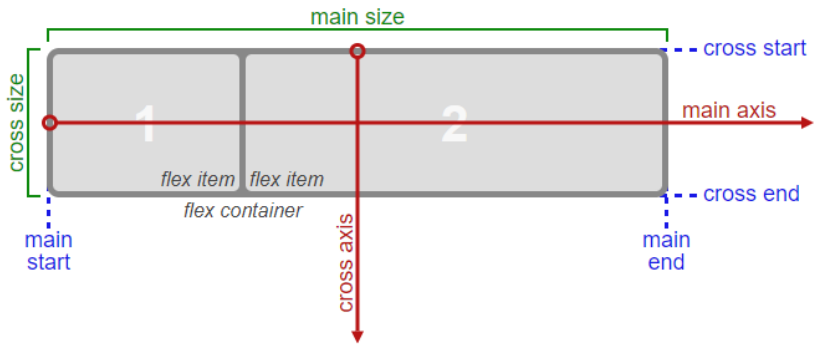
Flexible box layout[7] (flexibilní rozložení), často označovaný jako Flexbox[12], je modul CSS, jehož cílem je poskytnout efektivnější způsob rozložení, zarovnání a distribuci prostoru mezi položkami uvnitř kontejneru, i když velikost položek není známá nebo se jedná o dynamickou velikost (tedy právě slovo flex).

Jako hlavní myšlenku Flexbox layoutu lze považovat dání kontejneru možnost upravovat šířku či výšku (popřípadě pořadí) svých položek tak, aby co nejlépe vyplňovaly dostupné místo (většinou kvůli tomu, aby to ideálně vyhovovalo všem druhům zobrazovacích zařízení a velikostí obrazovek) uvnitř kontejneru.

Nejdůležitější věcí však je, že Flexbox layout je směrově agnostický (tj. bez jakýchkoliv směrových omezení) oproti běžným rozložením, které fungují dobře na stránkách, ale chybí jim flexibilita pro podporu velkých nebo složitých aplikací (zejména se jedná o změnu orientace nebo velikosti). Flexbox layout má v dnešních moderních prohlížečích téměř stoprocentní podporu, proto je vhodný k použití na novodobých webových stránek. A jak již bylo i zmíněno, s Grid layoutem se velmi dobře doplňují.

4.1 Základní terminologie

S Flexbox layoutem se také pojí několik nových termínů, které jsou důležité právě pro práci s tímto layoutem. Proto se tato kapitola bude těmito termíny zabývat. Patří sem hlavní a vedlejší osa, hlavní a vedlejší velikost a flex linky. Obrázek 4.1 znázorňuje některé termíny pro lepší ilustraci a přehlednost.



Obrázek 4.1: Ilustrace základních pojmů aplikovaných na řádkový flex kontejner. Obrázek je převzatý ze zdroje [3].

Hlavní a vedlejší osa

Hlavní (angl. main axis) a vedlejší (angl. cross axis) osa[3] určují uvnitř kontejneru směr, pomocí kterého budou jednotlivé položky rozkládány. Vedlejší osa je vždy kolmá vůči hlavní ose. Na základě těchto os se rozlišuje i výsledný kontejner, tedy jestli je horizontální nebo vertikální. Jak jde vidět i na obrázku 4.1, hlavní osa vždy v základním nastavení směřuje zleva doprava a vedlejší osa ze shora dolů. S hlavní osou se následně pojí i vlastnost CSS `flex-direction`.

Vlastnost flex-direction

Vlastnost `flex-direction` určuje, jakým způsobem jsou umisťovány flex položky uvnitř kontejneru na základě nastavení hlavní osy. Umístění flex položek se provádí pomocí následujících hodnot:

- row – položky jsou umisťovány základním způsobem, tedy zleva doprava
- row-reverse – stejný význam jako hodnota *row*, s tím rozdílem, že položky jsou umisťovány zprava doleva
- column – položky jsou umístěny vertikálním směrem, tedy ze shora dolů
- column-reverse – stejný význam jako hodnota *column*, s tím rozdílem, že položky jsou umisťovány zdola nahoru

Hlavní a vedlejší velikost

Hlavní (angl. main size) a vedlejší (angl. cross size) velikost[3] slouží jako velikosti pro obsah uvnitř flex kontejneru. Definují prostor, ve kterém se mohou rozkládat jednotlivé flex položky. Hlavní i vedlejší velikost lze nastavit klasickými vlastnostmi CSS, které se zabývají velikostmi. Pro hlavní velikost se především využívají vlastnosti *width*, *min-width* a *max-width*. Naopak pro vedlejší velikost se využívají vlastnosti *height*, *min-height* a *max-height*.

V případě horizontálního flex kontejneru (při vynechání hodnot pro velikost) je hlavní velikost nastavena na šířku prohlížeče a vedlejší velikost je počítána jako součet výšek jednotlivých flex linek. V případě vertikálního kontejneru je hlavní velikost nastavena na hodnotu nula a vedlejší velikost bude zastupovat šířka prohlížeče.

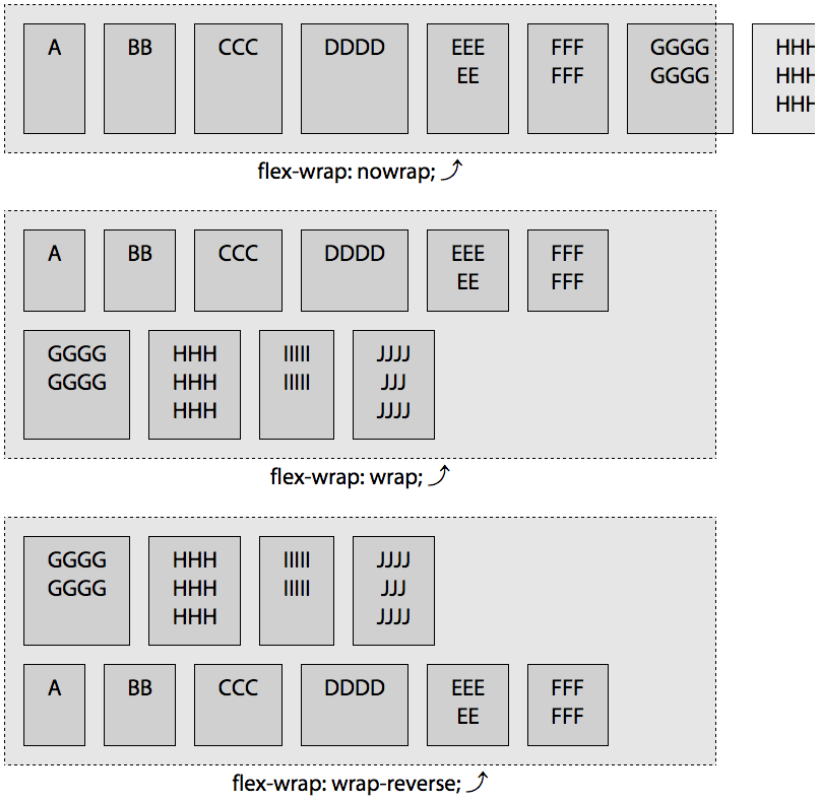
Flex linka

Jednotlivé flex položky jsou zarovnány a rozloženy ve flex linkách[3][11] (angl. flex lines). Flex linky představují hypotetické rozdělení kontejneru na oblasti, dále se využívají pro seskupení položek včetně jejich horizontálního a vertikálního zarovnání. Flex linky mohou obsahovat teoreticky nekonečně mnoho položek, podmínkou však je, aby v dané lince byla vždy minimálně jedna položka. Flex linku si lze představit jako jeden řádek uvnitř flexibilního kontejneru. S flex linkami se pojí i vlastnost CSS `flex-wrap`.

Vlastnost flex-wrap

Vlastnost `flex-wrap` určuje, zda flex kontejner bude jednořádkový (angl. single-line), nebo zda v případě potřeby se stane víceřádkovým (angl. multi-line). Pokud je tato vlastnost

nastavena tak, aby bylo povoleno více řádků, nové flex linky se objeví před nebo za původní flex linkou. Ve výchozím nastavení této vlastnosti jsou všechny flex položky umístěny pouze v jednom řádku, bez ohledu na jejich množství (můžou přesahovat na kontejner). Obrázek 4.2 shrnuje všechny hodnoty vlastnosti `flex-wrap` a ukazuje, jak se jednotlivé položky chovají na základě těchto hodnot.



Obrázek 4.2: Ukázka vlastnosti `flex-wrap` včetně hodnot podle kterých jsou položky umisťovány v kontejneru. Obrázek je převzatý ze zdroje [11].

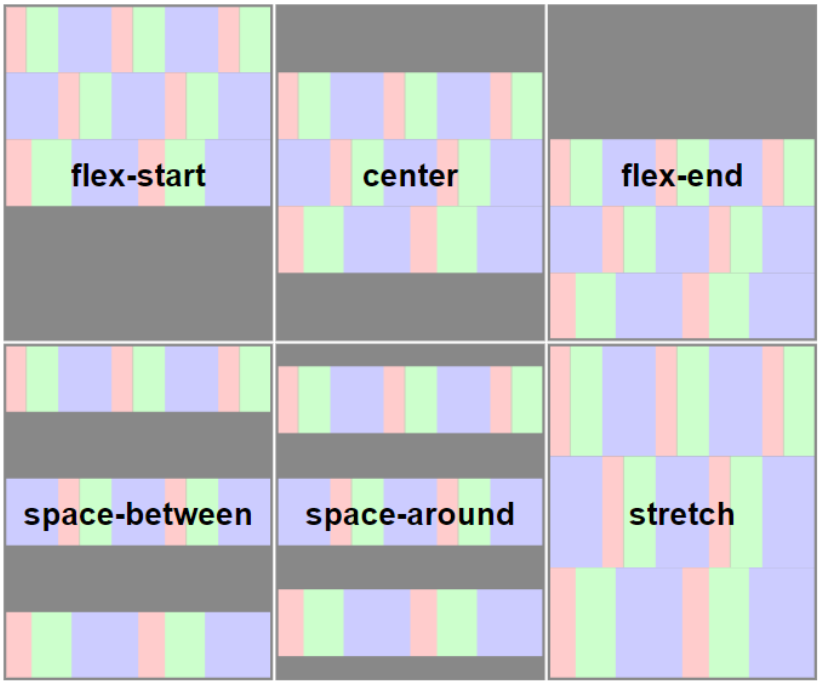
4.2 Flex kontejner

Flex kontejner[3] (angl. flex container) také s sebou přináší dvě nové hodnoty pro vlastnost *display* kaskádových stylů. Jedná se o hodnoty `display: flex` a `display: inline-flex`. První hodnota generuje flex kontejner, který se v rozložení stránky chová, jako by se jednalo o blokový element. Druhá hodnota rovněž generuje flex kontejner, ten se naopak chová jako inline element.

Stejně jako kontejner mřížky, tak i flex kontejner vytváří nový kontext formátování pro svůj obsah, tudíž místo blokového rozložení se využije flexibilní rozložení. Flex kontejner rovněž není považován za blokový kontejner, nemohou být na něj aplikovány vlastnosti, které jsou již shrnuty u kontejneru mřížky (viz. Kapitola 3.2). Závěrem je také dobré zmínit fakt, že pokud hodnota *display* flex kontejneru je nastavena na `display: inline-flex` a zároveň je tento kontejner plovoucí nebo absolutně umístěný, tak se hodnota vlastnosti *display* automaticky přepočítává na `display: flex`.

4.2.1 Zarovnání flex linek uvnitř kontejneru

Pro zarovnání flex linek se využívá vlastnost CSS `align-content`. Tato vlastnost určuje, jakým způsobem se bude volné místo distribuovat mezi linkami a kolem nich ve vedlejší velikosti kontejneru. Na jednořádkové kontejnery nemá vůbec žádný efekt, lze ji tedy použít pouze pro víceřádkové kontejnery. Oproti vlastnostem CSS `justify-content` a `align-items`, které se vztahují k jedné lince, se tato vlastnost vztahuje současně ke všem linkám kontejneru. Obrázek 4.3 shrnuje všechny hodnoty této vlastnosti a ukazuje, jaký vliv mají na položky v kontejneru.



Obrázek 4.3: Ilustrace klíčových hodnot vlastnosti `align-content` v horizontálním kontejneru. Obrázek je převzatý ze zdroje [3].

4.3 Flex položka

Flex položka[3] (angl. flex item) je reprezentována pomocí boxu, který představuje obsah daného kontejneru. Flex položka se také označuje jako potomek flex kontejneru. Pokud by položka byla zastoupena pouze textem bude následně obalena do tzv. annonymous boxu (vysvětlivka viz. poznámka pod čarou na straně 17). Do flex položky lze rovněž vkládat i další vnořené elementy, které mohou být maximálně tak velké, jako je daná flex položka. Flex položka může také obsahovat pouze bílé znaky, její chování bude vypadat tak, že obsah nebude zobrazen, ale položka jako taková se vykreslí.

4.3.1 Zjištění hlavní velikosti položek

Zjistit hlavní velikost položek je velmi důležité, protože se využívá k pozicování, vykreslování a hlavně pro nastavení hlavní velikosti vertikálního kontejneru. Položka potřebuje zjistit svůj tzv. hypotetický rozměr, který lze nastavit pomocí vlastnosti CSS `flex-basis` (dále

viz. Kapitola 4.3.3). Pokud tato vlastnost není nastavena (nebo obsahuje hodnoty *auto* či *content*), tak se pro hypotetický rozměr použijí vlastnosti `width` nebo `height` (v závislosti na typu kontejneru). Pokud však není nastavena ani šířka či výška, je použita vlastnost *min-content* (minimální hodnota obsahu). Před výsledným rozměrem položky je však potřeba ještě zpracovat tzv. flex faktory (viz. rovněž Kapitola 4.3.3), které mohou výsledný rozměr ještě poupravit.

4.3.2 Pozicování položek uvnitř kontejneru

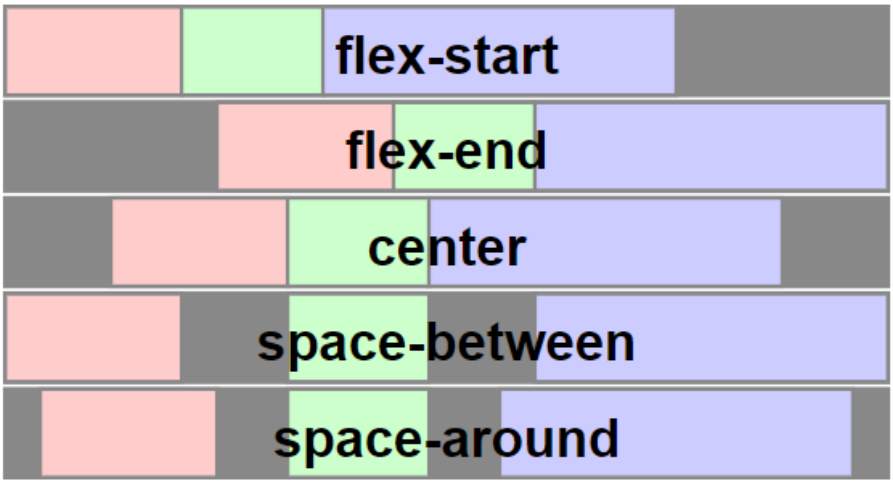
Již v základních pojmech bylo řečeno, že jsou položky v kontejneru umisťovány ihned za sebou. Zde byly probrány vlastnosti `flex-direction` a `flex-wrap`, tyto vlastnosti však provádějí s položkami pouze základní operace, které jsou pro náročnější pozicování nedostatečné. Pro náročnější pozicování položek vznikly vlastnosti CSS `order`, `justify-content` a `align-items` [11].

Vlastnost order

Vlastnost `order` se využívá ke změně řazení jednotlivých flex položek (v základním nastavení jsou ve stejném pořadí jako ve zdrojovém kódu). Ve výchozím nastavení jsou všechny položky řazeny od hodnoty nula. Pokud je potřeba přesunout položku na specifické místo, stačí ji nastavit hodnotu vlastnosti `order`.

Vlastnost justify-content

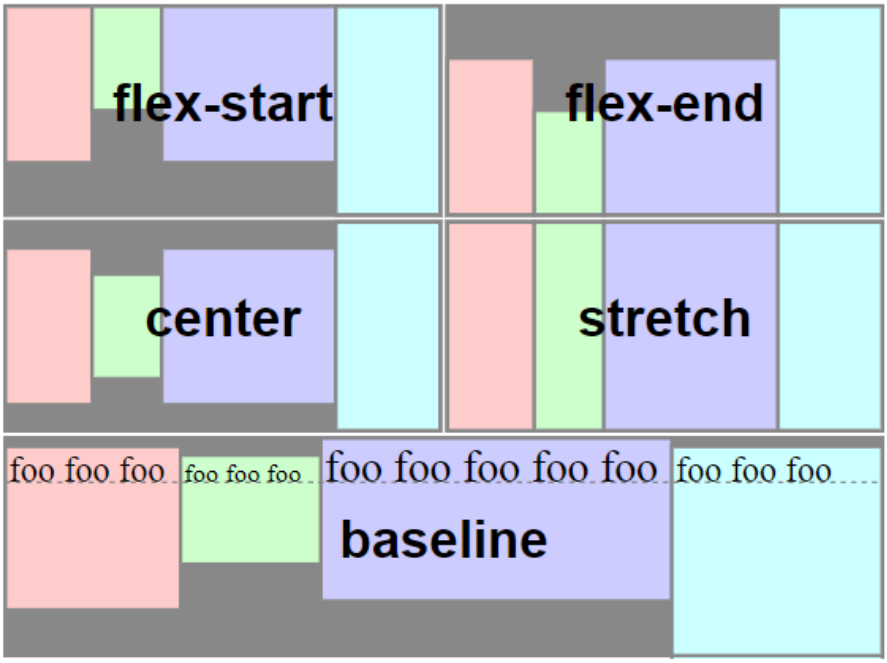
Vlastnost `justify-content` zarovnává položky podle hlavní osy v každém řádku kontejneru. Toto zarovnání se provede až po vyřešení flexibilních délek a automatických okrajů (vlastnost *margin*). Typicky tato vlastnost pomáhá distribuovat zbylé volné místo v kontejneru, když jsou všechny položky pevné nebo pružné, ale dosáhly své maximální velikosti. Rovněž má i kontrolu nad položkami, které přetékají mimo kontejner (určí, jakým směrem budou položky přetékat). Obrázek 4.4 shrnuje všechny hodnoty této vlastnosti a ukazuje, jaký vliv mají na položky v kontejneru.



Obrázek 4.4: Ilustrace klíčových hodnot vlastnosti `justify-content` v horizontálním kontejneru. Obrázek je převzatý ze zdroje [3].

Vlastnost align-items

Vlastnost `align-items` zarovnává položky podle vedlejší osy uvnitř kontejneru. Má podobné vlastnosti jako vlastnost `justify-content` a obě se aplikují na kontejner nikoliv na jednotlivé položky. Využívá i speciální hodnotu *baseline*, která může měnit výšku flex linky a zarovnává položky podle obsahu do roviny. Obrázek 4.5 shrnuje všechny hodnoty této vlastnosti a ukazuje, jaký vliv mají na položky v kontejneru.



Obrázek 4.5: Ilustrace klíčkových hodnot vlastnosti `align-items` v horizontálním kontejneru. Obrázek je převzatý ze zdroje [3].

4.3.3 Flexibilita položek

S flexibilitou položek se pojí tři vlastnosti CSS, kterými jsou `flex-grow`, `flex-shrink` a `flex-basis`. Za flexibilní položku[3] se považují ty položky, které mají nastavenou vlastnost CSS `flex-grow` nebo `flex-shrink`, v opačném případě se jedná o plně neflexibilní položku. Flexibilní položky se mohou roztahovat v dané lince nad rámec své velikosti, pokud hlavní rozměr linky obsahuje ještě nějaký volný prostor. Vlastnosti `flex-grow` a `flex-shrink` spolu dohromady označují tzv. flex faktory.

Vlastnost flex-grow

Vlastnost `flex-grow` definuje, zda má položka růst, když je k dispozici volný prostor. Pokud je k dispozici volný prostor, tato vlastnost určí, jak moc se položka roztáhne oproti ostatním položkám.

Vlastnost flex-shrink

Vlastnost `flex-shrink` určuje, o kolik se položka zmenší ve srovnání s ostatními položkami, když pro všechny není dostatek místa. Implicitní hodnotou této vlastnosti je hodnota jedna a nepřijímá záporné hodnoty.

Vlastnost flex-basis

Vlastnost `flex-basis` definuje počáteční (výchozí) velikost položky, předtím než bude položce umožněn růst nebo zmenšení podle flex faktorů.

Vlastnost flex

Vlastnost `flex` nepřináší nic nového, pouze spojuje předchozí vlastnosti. Používá se především kvůli přehlednosti kódu a navíc standard to i doporučuje, protože si lze na jednom místě specifikovat základní velikost položky a její flexibilitu. V CSS se tato vlastnost dá využít například následovně:

```
flex: 0 1 auto;
```

Kapitola 5

Knihovna CSSBox

Cílem této kapitoly je představení knihovny CSSBox. Popsat její strukturu, o jaké vstupy se jedná, co je výstupem a jak se provádí pozicování jednotlivých elementů. Je zde i krátce přestavena knihovna jStyleParser, která slouží jako analyzátor pro knihovnu CSSBox. Konec této kapitoly se věnuje popisu struktury knihovny CSSBox, která je rozšířená o mřížkové a flexibilní rozložení.

5.1 O knihovně

Knihovna CSSBox[4] je (X)HTML/CSS zobrazovací stroj, který je napsaný čistě v Javě. Mezi jeho primární účely patří poskytování kompletních a dále zpracovatelných informací o vykresleném obsahu stránky včetně rozložení. Nicméně lze tento zobrazovací stroj využít i pro prohlížení vykreslených dokumentů v Java swing aplikací.

Vstup zobrazovacího stroje představuje dokument, který je v podobě DOM stromu, a sada kaskádových stylů odkazovaných z daného dokumentu. Výstupem zobrazovacího stroje je objektově orientovaný model rozložení stránky. Tento model lze přímo zobrazit, avšak primárně je vhodný na další zpracování za pomocí algoritmů rozložení analýzy, mezi které lze zařadit například segmentace stránek nebo algoritmy extrakce informací.

Jádro knihovny CSSBox lze rovněž využít k získání bitmapového popřípadě vektorového (SVG) obrazu vykresleného dokumentu. Za pomoci balíčku SwingBox[1] lze využít knihovnu CSSBox jako interaktivní webový prohlížeč v aplikaci Java Swing.

Knihovna CSSBox je licencována za podmínek GNU Lesser General Public Licence version 3.

Zdrojové kódy knihovny jsou volně dostupné na serveru *GitHub*, konkrétně na této webové stránce[2]. U této knihovny je zaručena i kompatibilita na různých operačních systémech, které podporují jazyk Java.

5.2 Knihovna jStyleParser

Knihovna jStyleParser[5] je součástí projektu CSSBox a je rovněž napsaná v jazyce Java. Využívá se jako parser kaskádových stylů CSS. Obsahuje své aplikační rozhraní, které bylo navrženo tak, aby bylo umožněno efektivně zpracovat CSS v Javě a mapování těchto hodnot na datové typy, které Java poskytuje. Tato knihovna je také schopna aplikovat analyzované

[1] http://cssbox.sourceforge.net/swingbox/index.php
[2] https://github.com/radkovo/CSSBox

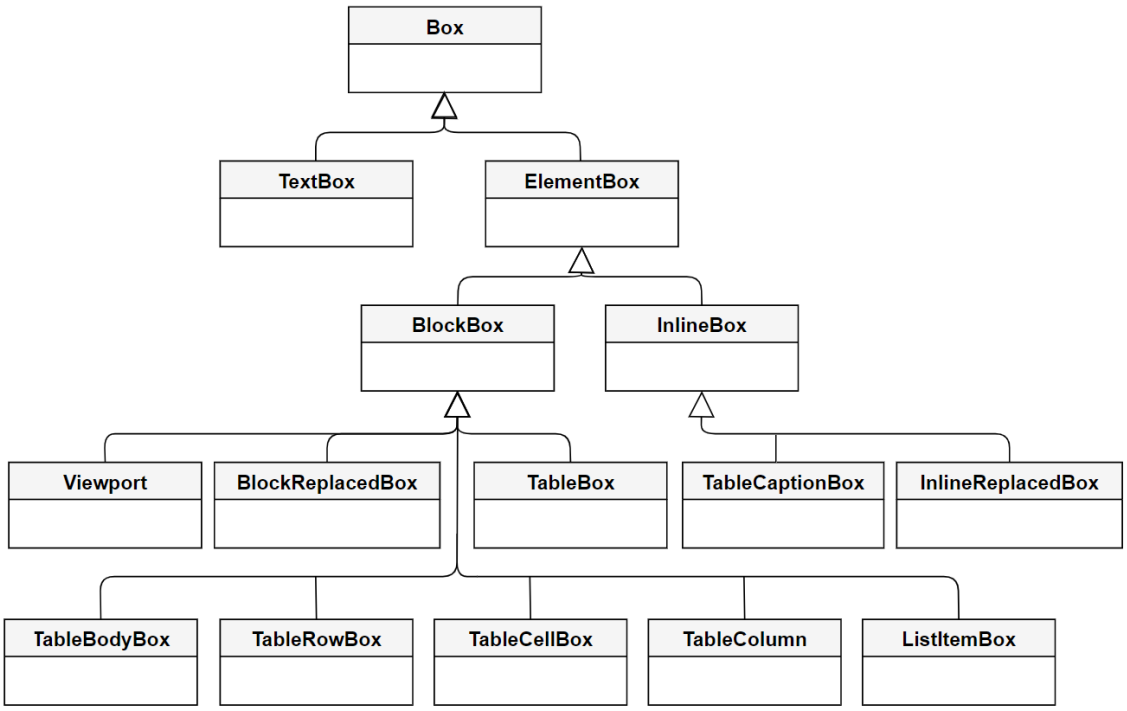
styly na DOM strom, reprezentující HTML případně XML dokument, a spočítat výsledný styl jednotlivých prvků dokumentu. Knihovna podporuje kaskádové styly podle specifikace CSS 2.1 a nově také i velkou podmnožinu specifikace CSS 3. CSSBox knihovna k těmto stylům dokáže přistupovat za pomoci tzv. Termlistů a tím si získat jejich hodnoty.

5.3 Vykreslený model dokumentu

Výsledný dokument či webová stránka jsou reprezentovány stromem boxů. Každý box ve stromu představuje obdélníkovou oblast ve výsledné stránce, která pak odpovídá konkrétnímu vykreslenému elementu jazyka HTML. Jeden element může představovat jeden box, který odpovídá nějaké části, nebo několik boxů, příkladem může být element `<p>`, ten pokud je víceřádkový, tak se vytvoří box pro odstavec a dále pro každý řádek textu.

Každý box je pak reprezentován objektem, rozšiřující abstraktní třídu `Box`. V původní struktuře knihovny CSSBox existovalo okolo deseti typů boxů, které zhruba odpovídají vlastnosti CSS `display` pro konkrétní element. Nyní je knihovna rozšířena o další dva typy boxů (viz. Kapitola 5.5). Na obrázku 5.1 je znázorněna původní hierarchie boxů, šipky na tomto obrázku znázorňují dědičnost mezi jednotlivými boxy.

Kořenový uzel stromu boxů vždy představuje objekt `Viewport`, ten přestavuje dostupnou plochu prohlížeče. Obsahuje vždy pouze jednoho potomka, kterému se říká kořenový box. Kořenový box pak odpovídá kořenovému elementu HTML (obvykle se jedná o element `<body>`) předaného objektu `BrowserCanvas` pro vykreslení.



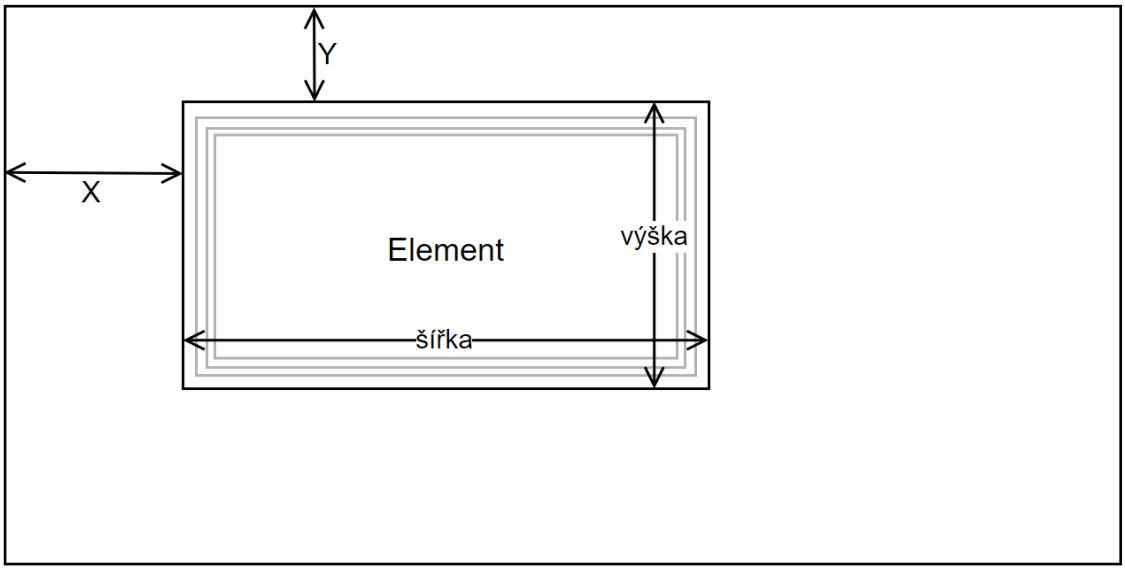
Obrázek 5.1: Původní hierarchie boxů v knihovně CSSBox. Tento obrázek byl upraven kvůli lepší čitelnosti a je převzat ze stránky[3] s oficiálním manuálem.

[3] http://cssbox.sourceforge.net/manual/

5.4 Pozicování v knihovně CSSBox

Pozicování v knihovně CSSBox vychází z již zmíněného CSS Box Modelu (viz. Kapitola 2.2). Pro pozicování elementů se využívá atributů, které jsou typu `Rectangle`. Jedná se o atributy `content` a `bounds`. Atribut `content` představuje čistě oblast pro obsah elementu, tedy bez vlastností CSS *border*, *margin* a *padding*. Atribut `bounds` v sobě zahrnuje oblast pro obsah elementu včetně již zmíněných vlastnosti CSS *border*, *margin* a *padding*. Na obrázku 5.2 jsou tyto dva atributy reprezentovány šířkou a výškou elementu, protože v sobě obsahují vnitřní atributy `width` a `height`, ty se nastavují pomocí metody `setSize(float width, float height)`. Pokud by šířka a výška nebyly nastaveny, element by se vykreslil s nulovou velikostí.

Dalšími vnitřními atributy jsou `x` a `y`, které jsou rovněž vyobrazeny na obrázku 5.2. Představují souřadnice pro nastavení pozice elementu, ta se nastavuje za pomocí metody `setLocation(float x, float y)`. Pozice pro umístění elementu se nastavuje vůči levému hornímu rohu svého rodičovského elementu. Pokud by byl element pozicován absolutně, pomocí vlastnosti CSS *position: absolute*, jeho pozice se bude nastavovat vůči levému hornímu rohu stránky.



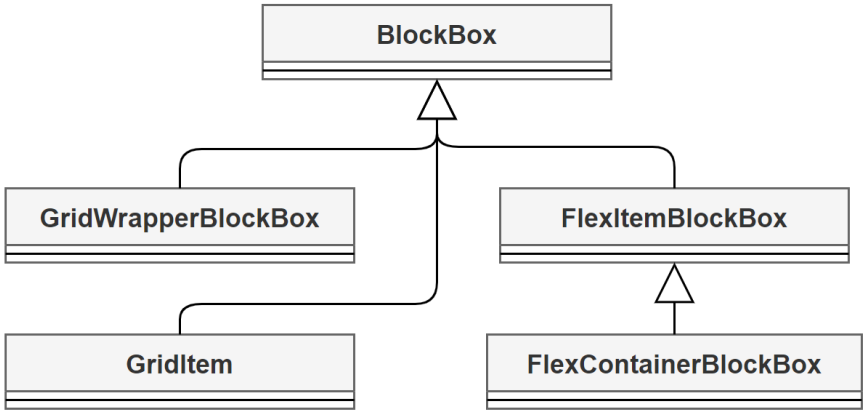
Obrázek 5.2: Pozicování elementu v knihovně CSSBox s atributy určující rozměry boxu.

5.5 Rozšířená struktura knihovny CSSBox pro Grid layout a Flexible box layout

Tato novější struktura byla vytvořena pro potřeby mřížkového a flexibilního rozložení, aby bylo možné následně s těmito rozloženími pracovat pomocí knihovny CSSBox. Strukturou se zabývaly dvě bakalářské práce, jedna se specializovala na mřížkové rozložení[14], druhá zase na flexibilní rozložení[15] a tzv. systém layout managerů byl společným tématem.

Nejprve byl vytvořen systém layout managerů, pomocí kterého lze boxům přidělit například jiný typ rozložení. Tento systém je dále popsán v kapitole 6.1, protože s ním tato práce bude rovněž pracovat.

Následně byly vytvořeny potřebné třídy pro fungování nových typů rozložení, které jsou znázorněny na obrázku 5.3. Tyto třídy převážně dědí ze třídy `BlockBox`, která je specifická pro blokové rozložení. Nicméně tento návrh se ve výsledné formě ukázal jako ne úplně správný, neboť bylo nutné modifikovat závislosti původního návrhu, zároveň nebyla možná integrace těchto nových rozložení a nebylo provedeno volání metody `doLayout()` na jednotlivé podboxy těchto rozložení.



Obrázek 5.3: Část digramu tříd znázorňující nově přidané třídy pro mřížkové a flexibilní rozložení. Třída `BlockBox` rovněž patří do hierarchie znázorněné na obrázku 5.1. Obrázek je vytvořený s pomocí bakalářských prací [14] a [15].

Kapitola 6

Návrh nové struktury a úprav knihovny CSSBox

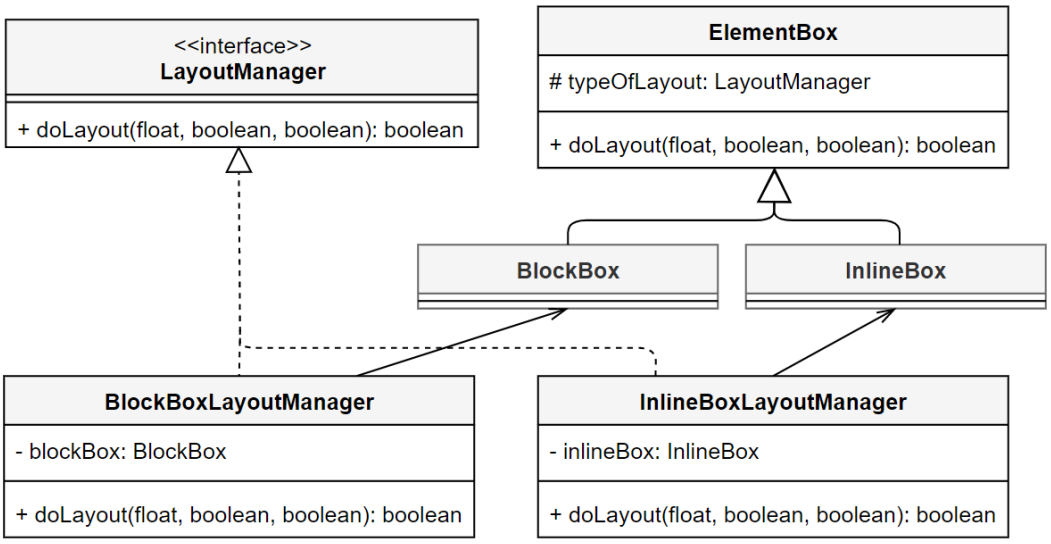
V této kapitole je přiblížen návrh nové struktury knihovny CSSBox. Ihned v úvodu je probrán systém layout managerů, který již byl experimentálně implementován v bakalářských pracích. Jsou zde popsány jeho výhody, za jakým účelem byl vytvořen a jaké kroky jsou potřeba pro jeho implementaci. V další kapitole je rozebráno začlenění nových typů rozložení do knihovny CSSBox. Přesněji kapitola pojednává o tom, jaké třídy pro nová rozložení je potřeba vytvořit v knihovně CSSBox včetně začlenění do systému layout managerů. Jsou zde i přiloženy diagramy tříd pro lepší ilustraci. Poslední část této kapitoly je věnována algoritmům pro nová rozložení, které jsou využity při implementaci v knihovně CSSBox.

6.1 Systém layout managerů v knihovně CSSBox

Jak již naznačila kapitola 5.5, původní struktura byla rozšířena právě o tento systém managerů. Tento systém byl navržen za účelem, aby se staral o strukturu rozložení hlavních boxů a zároveň bylo možné těmto boxům přidělit jiný způsob rozložení. Další výhodou tohoto systému je oddělení logiky zpracování rozložení od vnitřní logiky boxu, toto vede zároveň i k lépe přehlednému kódu.

V první řadě bylo navrženo rozhraní `LayoutManager`, které obsahuje popis metody `doLayout()`. Toto obecné rozhraní budou implementovat všechny konkrétní layout managery příslušných boxů.

Dále byla provedena úprava stávajících boxů (`BlockBox` a `InlineBox`), kdy byla přesunuta jejich metoda `doLayout()` do příslušných managerů. V managerech je navíc přidán atribut třídy kvůli identifikaci, pro který box byl vytvořen. Příslušné layout managery budou navíc ještě rozšířeny o další metody, které se týkají zpracování rozložení. Jinak řečeno, bude proveden potřebný refaktor kódu podle domluvy v již existujících boxech, tedy `BlockBox` a `InlineBox`. V nově přidaných boxech budou již příslušné metody v příslušných layout managerech. Celá struktura layout managerů s upravenými základními boxy je zobrazena na digramu tříd 6.1.



Obrázek 6.1: Diagram tříd znázorňující přidaný systém layout managerů s úpravou základních boxů.

Pro správné fungování managerů byl dále přidán atribut `typeOfLayout`, typu rozhraní `LayoutManager`, do abstraktní třídy `ElementBox`. Tento atribut je zde přidán z důvodu, aby k němu mohli přistupovat všichni potomci třídy `ElementBox`. Ti ho využívají již ve svém konstruktoru pro přiřazení konkrétního layout manageru. Ve třídě `ElementBox` byla ještě přidána metoda `doLayout()`, jejíž zápis je možné si prohlédnout ve výpisu 6.1. Metoda přiřadí konkrétní typ layout manageru a dále je zavolána, na základě daného typu boxu, metoda starající se o rozložení.

```
@Override
public boolean doLayout(float availw, boolean force, boolean linestart) {
        LayoutManager lm = typeoflayout;
        lm.doLayout(availw, force, linestart);
        return true;
}
```

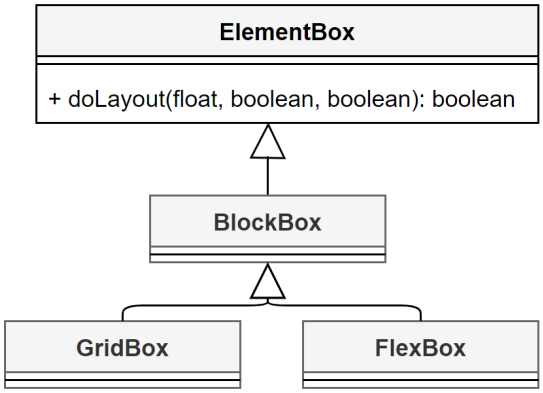
Výpis 6.1: Přidaná metoda `doLayout()` do abstraktní třídy `ElementBox`.

6.2 Začlenění Grid layoutu a Flexible box layoutu do knihovny CSSBox

Pro Grid layout a Flexbox byly nejprve navrženy dvě hlavní třídy, které budou představovat kontejnery daných rozložení. Jedná se o třídy `GridBox` a `FlexBox`, ty budou představovat potomky třídy `BlockBox` v hierarchii knihovny CSSBox (původně tyto nové třídy dědily z abstraktní třídy `ElementBox`, nicméně to se v pozdější fázi ukázalo jako nevhodné řešení). Tento vztah je vyobrazen na diagramu tříd 6.2.

Následující způsob začlenění byl zvolen z důvodu, že tyto třídy vytvářejí kontejnery pro své rozložení. Díky tomuto přístupu se tyto třídy vyskytují na stejné úrovni a mají přístup k metodám třídy `BlockBox`, které při své implementaci využívají. Na této úrovni

lze vytvářet i další nové kontejnery, pokud by bylo potřeba v budoucnu implementovat nové způsoby rozložení.

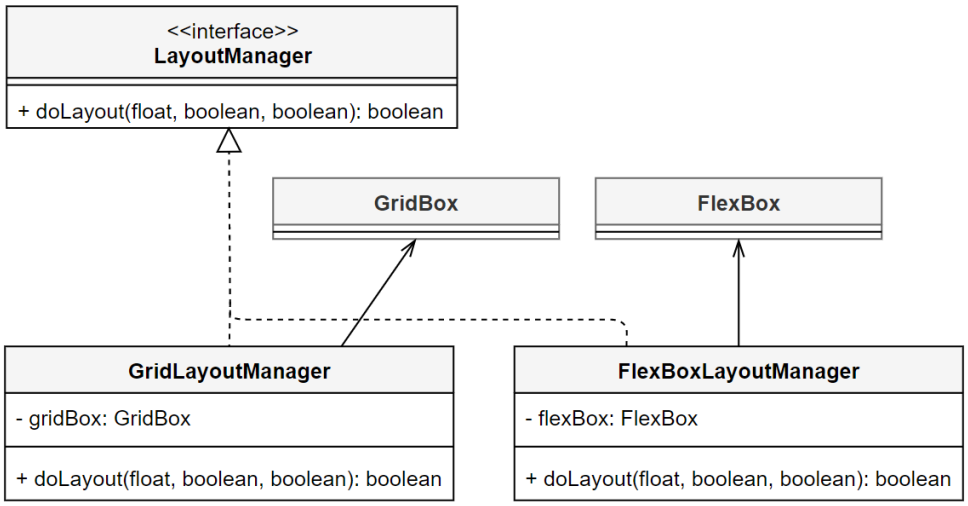


Obrázek 6.2: Diagram tříd znázorňující nově přidané boxy do hierarchie knihovny CSSBox.

Rovněž byly navrženy nové layout managery pro nová rozložení. Celý tento návrh je zobrazen diagramem tříd 6.3.

Byla navržena třída `GridLayoutManager` implementující obecné rozhraní `LayoutManager`, kde bude implementována metoda `doLayout()` podle Grid layout algoritmu a další potřebné metody pro rozložení. Do této třídy bude dále vložen atribut pro identifikaci příslušného mřížkového kontejneru.

Dále byla navržena třída `FlexBoxLayoutManager` stejným způsobem jako v případě mřížkového rozložení. Rovněž bude obsahovat daný atribut pro identifikaci flex kontejneru, implementovanou metodu `doLayout()` dle Flexbox algoritmu a další metody pro flexibilní rozložení.



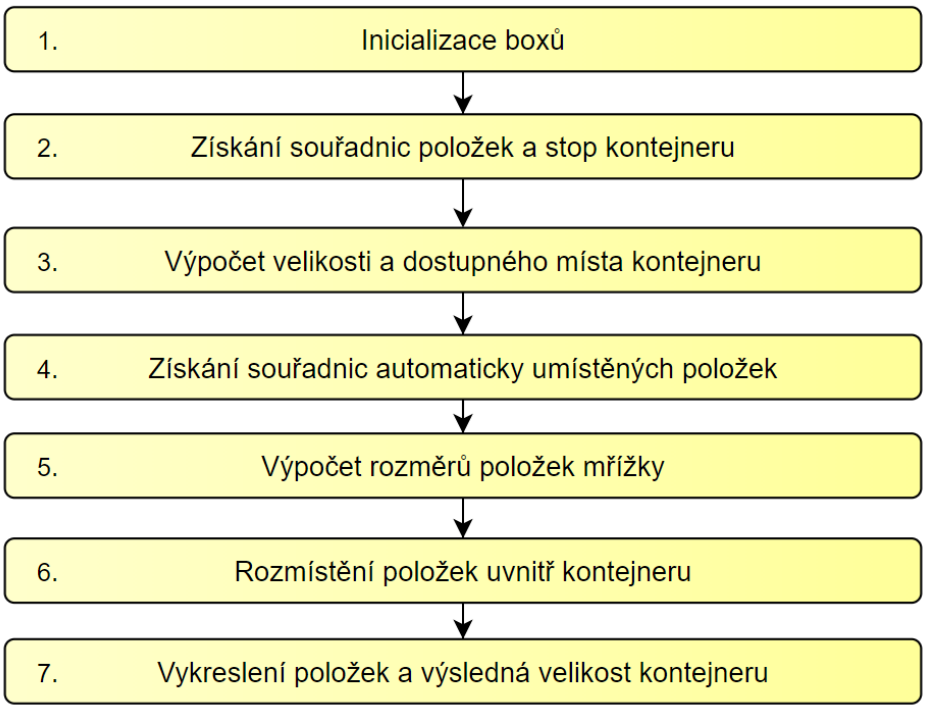
Obrázek 6.3: Diagram tříd zobrazující přidání systému layout managerů k nově vytvořeným typům boxů pro rozložení.

Jelikož dle specifikace CSS Grid layout a Flexbox pracují i ve svými položkami, které jsou přímými potomky příslušných kontejnerů, je potřeba s nimi počítat i v knihovně

CSSBox. Byly tedy navrženy třídy `GridItem` a `FlexItem`, které reprezentují příslušné položky. V těchto třídách se budou provádět všechny náležité operace, které s danými položkami souvisí.

6.3 Algoritmus Grid layout

V této sekci je vysvětlen algoritmus Grid layout, který je využit k implementaci mřížkového rozložení v knihovně CSSBox. Tento algoritmus je rozdělen do několika sekcí, které jsou zde stručně rozepsány pro lepší pochopení a jsou znázorněny na diagramu 6.4. Algoritmus si dále bere inspiraci z bakalářské práce[14] a z oficiálního algoritmu[1] od konsorcia W3C, který se zabývá obecným algoritmem Grid layout a jeho podrobným popisem.



Obrázek 6.4: Jednotlivé kroky algoritmu Grid layout, které jsou následně stručně rozepsány.

První krok algoritmu se zabývá správnou detekcí příslušných boxů mřížkového rozložení, je nutné dané boxy detekovat a vytvořit pro ně instance. V případě, že by přímý potomek kontejneru mřížky představoval `TextBox`, je nejprve potřeba tento box obalit anonymním boxem, jakmile se tak stane, pracuje se s ním dále jako s položkou mřížky. Dále se pak těmto boxům s pomocí knihovny `jStyleParser` získavají odpovídající vlastnosti stylů CSS. V tomto kroku se rovněž vytváří layout manager pro mřížkové rozložení.

Druhý krok algoritmu hledá, které položky mřížky mají pevné souřadnice umístění a pokud je najde, tak si zpracuje jejich hodnoty. Dále provede získání velikostí jednotlivých stop kontejneru.

[1] https://www.w3.org/TR/css-grid-1/#layout-algorithm

Třetí krok algoritmu pracuje s velikostí kontejneru, která je pak využita v pozdějších fázích. Také se zde řeší dostupný prostor v kontejneru, který je důležitý pro automaticky umístěné položky.

Čtvrtý krok algoritmu se zaměřuje na automaticky umístěné položky. Jelikož tyto položky nemají zadané pevné souřadnice, budou jim následně přiřazeny vzhledem k dostupnému místu uvnitř kontejneru mřížky.

Pátý krok algoritmu se zabývá výpočtem výšek a šířek jednotlivých položek mřížky.

Šestý krok algoritmu provádí výpočet vzdáleností jednotlivých položek vůči levému hornímu rohu kontejneru mřížky.

V posledním kroku algoritmu se provede kontrola výsledné velikosti kontejneru mřížky a dojde k vykreslení včetně jeho položek.

6.4 Algoritmus Flexible box layout

Tato sekce se naopak zabývá popisem algoritmu Flexible box layout, ten je využit pro implementaci flexibilního rozložení v knihovně CSSBox. Rovněž si bere inspiraci z bakalářské práce[15] a z oficiálního algoritmu[2] od konsorcia W3C, kde je podrobně popsaný. Algoritmus je rovněž stručně popsán pro lepší pochopení a jeho jednotlivé kroky jsou znázorněny na diagramu 6.5.

Obrázek 6.5: Jednotlivé kroky algoritmu Flexbox, které jsou následně stručně rozepsány.

[2] https://www.w3.org/TR/css-flexbox-1/#layout-algorithm

První krok algoritmu je zaměřený na detekci příslušných boxů flexibilního rozložení, jakmile jsou detekovány, vytvoří se pro ně odpovídající instance. Zde je také nutné, pokud by byl přímý potomek kontejneru `TextBox`, obalení do anonymního boxu. Rovněž se zde i získávají příslušné vlastnosti stylů CSS daných boxů za pomocí knihovny `jStyleParser`. V neposlední řadě je vytvořen i layout manager pro flexibilní rozložení.

Druhý krok algoritmu se zaměřuje na osy. Tedy je nutné zjistit, která osa bude hlavní osou. Na základě toho bude určen kontejner, zda je horizontální nebo vertikální (to určuje vlastnost `flex-direction`), a také se na této ose budou rozprostírat jednotlivé položky ve flex linkách. V tomto kroku je řešeno i pořadí položek podle vlastnosti `order` a další počáteční nastavení.

Třetí krok algoritmu počítá rozměry položek vůči vedlejší ose a dále se zabývá výpočtem hlavního hypotetického rozměru. V případě jsou-li zadány ohraničující minimální a maximální hodnoty, je potřeba tyto rozměry ohraničit těmito hodnotami.

Čtvrtý krok algoritmu pracuje s flex linkami, které v případě potřeby vytvoří. Podmínkou je, že daná flex linka musí mít minimálně jednu položku. Flex linkám je pak dále zvětšován jejich vedlejší rozměř na základě položky, která má největší vedlejší rozměr.

Pátý krok algoritmu provádí aplikaci flex faktorů na hlavní rozměr. Tedy, pokud je zadána vlastnost `flex-grow` a je nějaké volné místo v kontejneru, jsou položky roztaženy dle zadaných hodnot. V opačném případě, pokud je zadána vlastnost `flex-shrink` a položky přesahují hlavní rozměr, jsou smrsknuty dle zadaných hodnot.

Šestý krok algoritmu provede kontrolu, zda jsou položky inline elementy. Pokud tomu tak je, lze již vykreslovat jejich obsah. V opačném případě se jde na sedmý krok algoritmu.

Sedmý krok algoritmu umisťuje flex linky do kontejneru. V případě zadané vlastnosti `aling-content` a volného místa u vedlejšího rozměru kontejneru je nutné linky rozmístit na základě této vlastnosti. Pokud jsou ještě zadány vlastnosti pro zarovnání položek v linkách, je potřeba zpracovat i tyto vlastnosti.

Poslední krok algoritmu pouze nastavuje výslednou velikost flexibilního kontejneru.

Kapitola 7

Implementace mřížkového a flexibilního rozložení

Tato kapitola se z velké části zabývá implementací algoritmů pro mřížkové a flexibilní rozložení. Jsou zde podrobně popsány jednotlivé části algoritmů, které se v knihovně CSSBox odehrávají. U každé podkapitoly je vložen i krátký výpis části kódu, který se snaží vystihnout, co se v dané podkapitole odehrává. Konec této kapitoly se dále zabývá popisem dalších provedených změn po dohodě s vedoucím práce. Jsou zde představeny i návrhy na další rozšíření knihovny CSSBox a také odhalené nedostatky, které byly prokonzultovány a následně nahlášeny.

7.1 Mřížkové rozložení

Následující podkapitoly jsou věnovány podrobnému popisu algoritmu Grid layout, který již byl stručně rozebrán v kapitole 6.3.

7.1.1 Inicializace boxů

Správná funkčnost mřížkového rozložení v knihovně CSSBox je zaručena vytvářením adekvátních instancí příslušných boxů. Dané instance jsou tvořeny ve třídě `BoxFactory`. Pokud dojde k detekování elementu (kontejneru mřížky), který má vlastnost CSS `display: grid`, vytvoří se odpovídající instance třídy `GridBox`. Vytvoření dané instance je znázorněno ve výpisu 7.1. Tento výpis je součástí metody `createElementInstance()`, která se obecně stará o vytváření instancí třídy `ElementBox` na základě hodnoty vlastnosti *display*.

```
if(root.getDisplay() == ElementBox.DISPLAY_GRID)
        root = new GridBox((InlineBox) root);
```

Výpis 7.1: Vytvoření instance kontejneru pro mřížkové rozložení.

V této metodě je dále zachyceno vytváření instancí položek mřížky, jak uvádí výpis 7.2. Jakmile daný box představuje přímého potomka kontejneru mřížky, vytvoří se instance třídy `GridBox`.

```
if (parent instanceof GridBox)
root = new GridItem((InlineBox) root);
```

Výpis 7.2: Vytvoření instance položky mřížky.

Pokud by nastala situace, že čistý textový box by měl být položkou mřížky, je nutné tento box nejprve obalit do anonymního textového boxu (vlastnosti daného textového boxu jsou předány anonymnímu boxu), a poté se z něj stane položka mřížky. Tento princip je vyobrazen ve výpisu 7.3 a nachází se v metodě `createAnonymousBox()`, která se obecně stará o anonymní elementy.

```
if (parent.display == ElementBox.DISPLAY_GRID) {
    anbox = new GridItem(anelem, child.getVisualContext().create());
    anbox.setStyle(createAnonymousStyle("block"));
```

Výpis 7.3: Vytvoření instance anonymní položky mřížky.

Dále je pak v příslušném konstruktoru mřížkového kontejneru vytvořena instance layout manageru, kdy se jedná o `GridLayoutManager`. Tato instance je následně uložena v atributu `typeOfLayout`.

Styly CSS pro boxy mřížkového rozložení

Nyní je potřeba načíst všechny dostupné styly CSS pro příslušné boxy mřížkového rozložení. K této operaci je využita metoda `setStyle()`, která se stará o přiřazování nového stylu k danému boxu. Ta funguje tak, že hierarchicky načítá všechny styly daného boxu a dalších boxů, ze kterých dědí (jde o třídy `BlockBox` a `ElementBox`).

Ke zpracování konkrétního stylu byla následně využita knihovna `jStyleParser`, přesněji se jedná o konstrukci `style.getProperty("zadany-styl")`. Dále se pak zjišťuje hodnota daného stylu (jestli se jedná například o číselný údaj, procenta, řetězec, minimální hodnotu apod.). V případě další potřeby (například pro číselné hodnoty) k získání skutečné hodnoty stylu je využito instance třídy `CSSDecoder`. Ta obsahuje metodu `getLength()`, která načte hodnoty a následně je převádí na pixelové jednotky.

Všechny načtené hodnoty stylů jsou uloženy v příslušných atributech konkrétních tříd. V případě, že není zadán nějaký styl, je mu automaticky přidělena jeho výchozí hodnota, kterou lze najít v oficiální specifikaci CSS pro mřížkové rozložení.

7.1.2 Souřadnice položek a stopy kontejneru

K uchování souřadnic jednotlivých položek mřížky byla využita třída `GridItemCoords`. Tato třída obsahuje čtyři základní atributy pro uchování souřadnic, ty představují počátek řádku či sloupce a dále také konec řádku či sloupce. Souřadnice položek jsou reprezentovány číselnými jednotkami. Avšak ne vždy to tak je, občas mohou mít položky nastaveno roztahování o několik řádků či sloupců. Kvůli tomuto případu je třída `GridItemCoords` rozšířena o další čtyři atributy, které zastupují roztažení položky. Také může nastat situace, kdy položky nemají žádné souřadnice, nebo jsou nastaveny na hodnotu *auto*, v této situaci jsou brány jako automaticky umístěné položky (jejich souřadnice jsou nastaveny na hodnotu nula) a jsou zpracovány v jiném kroku algoritmu. Proces zjištění a uložení souřadnice je znázorněn ve výpisu 7.4. Tento výpis se zabývá počáteční souřadnicí, nicméně pro ostatní souřadnice je proces obdobný.

```
if (gridStart == null) gridStart = CSSProperty.GridStartEnd.AUTO;

if (gridStart == CSSProperty.GridStartEnd.number) {
        gridItemCoords.rowStart =
            dec.getLength(getLengthValue("grid-row-start"), ...);
} else if (gridStart == CSSProperty.GridStartEnd.AUTO) {
        gridItemCoords.rowStart = 0;
} else if (gridStart == CSSProperty.GridStartEnd.component_values) {
        gridItemCoords.rowStartSpan = style.getValue(TermList.class,
            "grid-row-start");
}
```

Výpis 7.4: Zjištění počáteční souřadnice řádku položky mřížky na základě zadané vlastnosti `grid-row-start`. Hned na začátku se provádí kontrola, zda je vlastnost zadána či nikoliv.

Pokud má položka nastaveno roztažení, je ještě nutné z daného `TermListu`[1] na první pozici vzít hodnotu roztažení. Jedná-li se o koncové roztažení, je tato hodnota přičtena k počáteční souřadnici a uložena jako koncová. V opačném případě je počáteční hodnota roztažení odečtena od koncové souřadnice a uložena jako počáteční.

Stopy kontejneru

Explicitní mřížka kontejneru je ukládána již do zmíněného *TermListu*, viz. výpis 7.5. Díky tomu lze ke každé stopě přistupovat pomocí dotazu na index. Jakmile je znám potřebný index, začne se zpracovávat hodnota dané stopy. Pokud stopa obsahuje číselnou hodnotu, je zpracována již zmíněnou metodou `getLength()`. Pokud se ve stopách vyskytují hodnoty, jako jsou například *min-content*, *max-content* apod., je nutné pracovat s obsahovou částí položek, tím se zabývají další kroky algoritmu.

```
if (gridTemplateRows == CSSProperty.GridTemplateRowsColumns.list_values) {
        TermList gridTemplateRowValues = style.getValue(TermList.class,
            "grid-template-rows");
}
```

Výpis 7.5: Uložení všech stop explicitní mřížky kontejneru do struktury *TermListu*.

Implicitní mřížka kontejneru již není složena z několika definovaných stop, obsahuje tedy pouze jednu hodnotu, stejnou pro všechny stopy implicitní mřížky. Implicitní mřížka může rovněž obsahovat číselné jednotky nebo nečíselné názvy. V případě číselných hodnot, jsou klasicky zpracovány metodou `getLength()`. Nečíselné názvy jsou zastoupeny odpovídajícími atributy boolovského typu a za základě nich jsou počítány adekvátní velikosti stop.

Jednotka fr

Jelikož jednotka fr přišla až s mřížkovým rozložením, není aktuálně podporována v knihovně CSSBox. Pro její podporu je nutné provést odpovídající změny v abstraktní třídě `VisualContext`. Přesněji se jedná o metodu `pxLength()`, která převádí různé jednotky do pixelů.

[1] *TermList* obsahuje seznam pojmů (např. 200px min-content 50px), ke kterým umožňuje přístup a je z knihovny `jStyleParser`

V této metodě se vyskytuje konstrukce *switch*, do které byl přidán nový případ zahrnující jednotku fr a tento případ vrací hodnotu jednotky fr.

Co se týká samotné implementace, je nutné nejprve najít všechny jednotky fr vyskytující se ve stopách kontejneru mřížky (ty se následně sečtou). Následuje kontrola stop, které brání vyplnění prostoru (jedná se například o stopy s pixelovými jednotkami). Tyto stopy (včetně vlastností `border`, `margin` a `padding`) jsou poté odečteny od celkové velikosti kontejneru ke zjištění zbylého volného prostoru. Nyní stačí tento volný prostor vydělit celkovým počtem jednotek fr a je zjištěna velikost *1fr* v pixelových jednotkách, která je uložena v příslušném atributu kontejneru. Základem pro tento postup je vzorec 3.1 z kapitoly 3.1.

7.1.3 Velikost a dostupné místo kontejneru

Základní kontejner mřížky je reprezentován velikostí 1x1 (tedy, že se do něj vejde jedna položka mřížky). Velikost kontejneru představují atributy `maxRowLine` a `maxColumnLine`. Tyto atributy se dají považovat za pomocné linky mřížky, které již byly probrány v teoretické části. Nicméně tato základní velikost ve většině případů není finální, je tedy nutné zjistit aktuální hypotetickou velikost kontejneru mřížky.

To se postupně provádí zjišťováním, jaké položky uvnitř sebe obsahuje. Aktuálně jsou ignorovány všechny automaticky umístěné položky, protože zatím neznají svoje souřadnice, kam by se mohly umístit. Prohledávají se tedy položky, které mají nastaveny svoje souřadnice pro umístění v kontejneru z předchozího kroku a jsou zjišťovány jejich koncové souřadnice, ze kterých jsou následně vybrány ty nejvyšší, zastupující hypotetický největší rozměr kontejneru. Tento zjednodušený princip je ukázán ve výpisu 7.6.

```
if (griditem.gridItemCoords.columnEnd > maxColumnLine) {
        maxColumnLine = griditem.gridItemCoords.columnEnd;
}

if (griditem.gridItemCoords.rowEnd > maxRowLine) {
                maxRowLine = griditem.gridItemCoords.rowEnd;
}
```

Výpis 7.6: Zjednodušený výňatek nastavování hypotetického největšího rozměru kontejneru mřížky.

Jelikož máme implicitní a explicitní mřížku, dost často nastane jejich kombinace v kontejneru. Je proto nutné zjistit ještě směr umisťování položek mřížky podle vlastnosti `grid-auto-flow`. Na základě toho je příslušný atribut velikosti kontejneru inkrementován o jedna.

Je nutné také počítat s případem, kdy v kontejneru budou umístěny pouze automatické položky. Zde opět hraje důležitou roli vlastnost `grid-auto-flow` určující směr položek. Tento efekt je obdobný jako flex linka ve flexboxu. Do jednoho atributu velikosti kontejneru je nahrána hodnota představující počet položek mřížky inkrementován o jedna, druhý atribut zůstává v základní podobě, tedy s hodnotou jedna.

Ke zpracování dostupného místa uvnitř kontejneru je využito struktury `ArrayListu`, která je postupně naplňována hodnotami v cyklu do maximální velikosti kontejneru mřížky. Toto naplnění se provádí pro každý řádek či sloupec v závislosti na vlastnosti `grid-auto-flow`. Tento proces je znázorněn v první části výpisu 7.7 a naplněné hodnoty představují budoucí počáteční souřadnice automatických položek mřížky.

```
for (int a = 1; a < maxRowLine; a++) {
    ArrayList<Integer> columnID = new ArrayList<>();
    for (int b = 1; b < maxColumnLine; b++)
        columnID.add(b);

    for (int i = 0; i < getSubBoxNumber(); i++) {
        GridItem griditem = (GridItem) getSubBox(i);
        if (griditem.gridItemCoords.rowStart == a) {
            for (int c = griditem.gridItemCoords.rowStart; c <
                griditem.gridItemCoords.columnEnd; c++)
                columnID.remove((Integer) c);
        }
    }
}
```

Výpis 7.7: Naplnění `ArrayListu` v závislosti na velikosti kontejneru, poté následuje odstraňování položek mřížky s pevnými souřadnicemi, které jsou rozprostřeny v jedné oblasti mřížky.

Po naplnění hodnot je poté nutné odstranit všechny hodnoty z `ArrayListu`, které zastupují již předem známé souřadnice pevných položek mřížky. Tento princip je znázorněn ve druhé části výpisu 7.7, kdy dochází k odstraňování hodnot položek, které jsou pouze v jedné oblasti mřížky. Následuje odstranění hodnot zastupující položky rozpoložené do více oblastí, kdy je důležité provádět i zpětnou kontrolu vůči jejich předešlým souřadnicím. Když nebude prováděna tato kontrola, mohlo by se stát, že dojde k odstranění špatné hodnoty. Výsledný dostupný prostor je pak využit v následujícím kroku.

7.1.4 Automaticky umístěné položky

Automaticky umístěné položky na základě dostupného prostoru kontejneru mřížky se mohou rozkládat pouze v jedné oblasti mřížky. Je to způsobeno v důsledku toho, že neznají svoje souřadnice a dle specifikace se tak chovají. Souřadnice těchto položek se budou rovněž ukládat do pomocné třídy `GridItemCoords`. Pokud by položek bylo více, než je aktuální hypotetická velikost kontejneru, je velikost automaticky dle potřeby rozšířena o nový řádek či sloupec.

Je zde potřeba také znát hodnotu vlastnosti `grid-auto-flow`, nicméně ta je známa z již předchozího kroku, protože tento krok přímo navazuje na ten předchozí. Na základě této hodnoty je přidělena počáteční souřadnice položky aktuálního řádku nebo sloupce. Následně je dopočítána koncová souřadnice, kdy se pouze inkrementuje počáteční souřadnice o jedna a je uložena do příslušného atributu. V případě, že byla zadána orientace pro řádek (nebo naopak pro sloupec) je samozřejmě nutné dopočítat i zbylé 2 souřadnice. Zbylou počáteční souřadnici lze zjistit z již zmíněného `ArrayListu` v předchozím kroku, ta se nachází na nulté pozici v tomto listu. Jakmile je hodnota získána, lze díky ní dopočítat i koncovou souřadnici, kdy se zase inkrementuje počáteční o jedna. Princip přiřazování všech souřadnic automatickým položkám je nastíněn ve výpisu 7.8. Nyní jsou již známy všechny čtyři souřadnice daných položek.

```
 for (int w = 0; w < getSubBoxNumber(); w++) {
        GridItem griditem = (GridItem) getSubBox(w);
        if (griditem.gridItemCoords.rowStart == 0 &&
            griditem.gridItemCoords.columnStart == 0) {
                if (columnID.isEmpty()) break;
                griditem.gridItemCoords.rowStart = a;
                griditem.gridItemCoords.rowEnd = a + 1;
                griditem.gridItemCoords.columnStart = columnID.get(0);
                griditem.gridItemCoords.columnEnd =
                    griditem.gridItemCoords.columnStart + 1;
                columnID.remove(0);
        }
}
```

Výpis 7.8: Princip přiřazování nových souřadnic automaticky umístěných položek na základě dostupného místa z předchozího kroku.

V poslední části se provádí kontrola přeplnění kontejneru. Pokud se zde vyskytují ještě položky bez svých souřadnic, je rozšířena hypotetická velikost kontejneru a proces přiřazování souřadnic položkám je opakován.

7.1.5 Výpočet rozměrů položek

Nyní již lze přistoupit k výpočtu rozměrů jednotlivých položek, protože položky v tomto kroku už znají své souřadnice. Celý proces výpočtu rozměrů jednotlivých položek je prováděn v cyklu, který se odvíjí od počáteční a koncové souřadnice dané položky, jak již naznačuje výpis 7.9. Než začne samotné počítání rozměrů položek, je nutné řešit, jaké máme aktivní mřížky a ve které se zrovna nacházíme.

Pokud máme aktivní pouze implicitní nebo explicitní mřížku, velikosti stop, přes které se položka roztahuje, jsou přičítány do aktuálních rozměrů položky.

V případě obou aktivních mřížek se nejdříve provádí výpočet velikosti explicitní mřížky. Jednotlivé stopy explicitní mřížky jsou uloženy v `TermListu`, který zároveň bude představovat hranici. Aktuálně se na základě jednotlivých souřadnic bude zjišťovat, ve které mřížce se položka nachází. Jestliže se souřadnice vyskytují v explicitní mřížce, je počítána velikost položek vůči stopám, které jsou získané z `TermListu`. Pokud jsou položky mimo explicitní mřížku, jejich velikost již zastupuje stopa implicitní mřížky.

Je také nutné počítat s případy, kdy se položky roztahují přes více než jednu buňku mřížky. V tomto případě je nutné počítat i s mezerami mezi stopami, které se musejí také zohlednit ve výsledné velikosti položek. Princip zohlednění mezer mezi stopami je poukázán v posledním řádku výpisu 7.9, kdy se od koncové souřadnice položky odečte počáteční. Následně dojde k dekrementaci o jedna a poté je hodnota vynásobena velikostí dané mezery.

```
for(int j=gridItemCoords.columnStart-1; j<gridItemCoords.columnEnd-1; j++)
{
  switch (gridBox.gridTemplateColumnValues.get(gridItemCoords.columnStart
     - 1).getValue().toString()) {
       case "min-content":
         widthOfGridItem = getMinimalWidth();
         break;
```

```
        case "max-content":
          widthOfGridItem = getMaximalWidth();
          break;
        case "auto":
          widthOfGridItem = 0;
          break;
  }
}
widthOfGridItem += (gridItemCoords.columnEnd - gridItemCoords.columnStart
    - 1) * gridBox.gapColumn;
```

Výpis 7.9: Ukázka zjednodušeného principu přidělování šířky položky na základě hodnot `min-content` a `max-content`, hodnota `auto` jde zde dočasně nastavena na nulovou velikost.

Jednotlivé velikosti stop mřížky mohou představovat i zadané hodnoty `auto`, `min-content` a `max-content`. Řešení těchto hodnot je nastíněno ve výpisu 7.9, kdy se pomocí konstrukce `switch` vybírá, o jakou hodnotu zrovna jde. V případě hodnoty `min-content` je využívána metoda `getMinimalWidth()` (obdobně to platí i pro výšku), která určuje minimální šířku boxu. V případě hodnoty `max-content` je naopak využita metoda `getMaximalWidth()` (obdobně to platí i pro výšku), která určuje maximální šířku boxu na základě jeho obsahu. Hodnota `auto` je momentálně nastavena na hodnotu nula a je zpracována později.

Díky těmto hodnotám mají jednotlivé položky rozdílné hodnoty, což není žádoucí a je nutné, aby položky v daném sloupci či řádku měly stejné velikosti. Je proto nutné znovu projít příslušné položky a donastavit jim správnou velikost. Jen pro upřesnění, pokud se jedná o řádkové stopy, tak dle specifikace hodnoty `auto` a `max-content` vykazují stejné chování jako hodnota `min-content`.

$$auto\_unit = \frac{available\_space - grid\_gaps - sum\_of\_non\_auto\_tracks}{count\_of\_auto\_tracks} \tag{7.1}$$

Nyní se lze zabývat hodnotou `auto`. Je nutné sečíst všechny velikosti stop, které nabývají jinou hodnotu, než je hodnota `auto`. Dále je nutné prověřit a spočítat počet výskytu hodnoty `auto`. Jakmile máme zjištěné tyto dva údaje, jsou ještě spočítány mezery mezi jednotlivými stopami, které zde také hrají svojí roli. V tomto bodě již lze vycházet ze vzorce 7.1, kdy se od dostupného prostoru kontejneru odečtou mezery mezi stopami s velikostmi stop (bez hodnoty `auto`). Celý výsledek se poté vydělí počtem výskytů hodnoty `auto` a je dostupný potřebný výsledek. Výslednou hodnotu je nutné kontrolovat i vůči minimálnímu obsahu položky, protože se může stát, že mohou vycházet i velmi malé hodnoty (nebo záporné), pokud tato situace nastane, je velikost položky nastavena na minimální velikost podle obsahu.

7.1.6 Rozmístění položek v kontejneru

Tento krok se zabývá nastavováním atributů `x` a `y`, ty zastupují umístění boxů od levého horního rohu. K finálnímu nastavení těchto atributů je využita metoda `setPositiion()`. Celý proces je rovněž prováděn v cyklu, kdy ukončovací podmínka cyklu je nastavena na počáteční souřadnici odpovídající položky. Než začne samotné počítání vzdáleností, je také nutné řešit, jaké máme aktivní mřížky a ve které se zrovna nacházíme.

Pokud je aktivní pouze explicitní nebo implicitní mřížka, jsou velikosti jednotlivých stop jednoduše přičítány do příslušných atributů, to již nastiňuje zjednodušeným způsobem výpis 7.10. Příslušné atributy se později přenastaví na atributy `x` a `y`.

V případě aktivních obou mřížek je nejprve zjištěna velikost explicitní mřížky. Nyní záleží na jednotlivých souřadnicích položek, díky nim lze zjistit, ve které mřížce se nacházejí. Pokud se počáteční souřadnice položek vyskytují v explicitní mřížce, je zase jednoduše spočítána jejich vzdálenost vůči jednotlivým stopám mřížky. Jakmile jsou už položky detekovány mimo, je nejprve spočtena velikost explicitní mřížky, která je ještě dále rozšířena o další vzdálenost udávající, jak moc daleko se daná položka nachází v implicitní mřížce.

```
for (int j = 0; j < gridItemCoords.columnStart - 1; j++) {
        if (gridItemCoords.columnStart == 1) {
                columnDistanceFromZero = 0;
        } else {
                columnDistanceFromZero += gridBox.arrayofcolumns.get(j);
        }
}
columnDistanceFromZero += (gridItemCoords.columnStart-1)*gridBox.gapColumn;
```

Výpis 7.10: Zjednodušený princip nastavování položkám jejich vzdálenost od levého horního rohu.

Nyní již jsou známy jednotlivé vzdálenosti, je tedy nutné také zohlednit mezery mezi jednotlivými stopami, se kterými se musí rovněž počítat. Princip spočítání těchto mezer je znázorněn v posledním řádku výpisu 7.10, kdy se jednoduchým způsobem vynásobí velikost dané mezery vůči počáteční souřadnici příslušné položky mřížky.

7.1.7 Vykreslení položek a výsledná velikost kontejneru

K vykreslování položek (zobrazení obsahu) se obecně používá metoda `doLayout()` z abstraktní třídy `ElementBox`. Tato metoda obsahuje tři parametry, přičemž první parametr představuje, s jakou předanou šířkou dané položky má počítat. Metodě je tedy předán atribut `content.width`, který zastupuje již předem spočítanou šířkou dané položky mřížky. Díky tomuto způsobu se již nemusí zasahovat do třídy `BlockBoxLayoutManager` v podobě upřesnění, zda se jedná či nejedná o položky mřížky.

Výsledná velikost kontejneru mřížky

Posledním krokem Grid layout algoritmu je zajištění správného zobrazení kontejneru mřížky v knihovně CSSBox. Zde mohou ještě figurovat vlastnosti ohledně výšky, jako jsou `height`, `max-height` případně `min-height`.

Při zadané vlastnosti `height` je to jednoduché, akorát se vezme její hodnota, která bude představovat finální velikost kontejneru. V případě ostatních hodnot je nutné kontrolovat, zda aktuální velikost kontejneru nepřesahuje zvolené ohraničující hodnoty. Pokud by byly přesaženy tyto hodnoty, kontejneru je přenastavena maximální nebo minimální výška, podle zadané vlastnosti. Velikost ještě může být v některých případech kontrolována vůči `ArrayListu`.

Kontejneru jsou poté nastaveny jeho atributy `bounds` a `content`. Následně je ještě provedena korekce s vlastnostmi `border`, `margin` a `padding`.

7.2 Flexibilní rozložení

Následující podkapitoly se budou zabývat podrobným popisem algoritmu Flexbox, který již byl představen a nastíněn v kapitole 6.4.

7.2.1 Inicializace boxů

Pro správnou funkčnost flexibilního rozložení v knihovně CSSBox je rovněž nutné nejprve vytvářet správné instance příslušných boxů. Vytváření instancí daných boxů, jak již bylo řečeno, je prováděno ve třídě `BoxFactory`. Jakmile bude detekován element s vlastností CSS `display: flex`, je pro něj vytvořena instance třídy `FlexBox`. Samotné vytvoření kontejneru lze vidět ve výpisu 7.11. Tato část se také nachází v metodě `createElementInstance()`.

```
if(root.getDisplay() == ElementBox.DISPLAY_FLEX)
        root = new FlexBox((InlineBox) root);
```

Výpis 7.11: Vytvoření instance kontejneru pro flexibilní rozložení.

Ve stejné metodě je ještě prováděno vytváření instancí pro flexibilní položky, to je znázorněno ve výpisu 7.12. Pokud by daný box byl přímým potomkem flexibilního kontejneru, je pro něj vytvořena instance třídy `FlexItem`.

```
if (parent instanceof FlexBox)
        root = new FlexItem((InlineBox) root);
```

Výpis 7.12: Vytvoření instance flexibilní položky.

Je také nutné odchytit případ, kdy se má stát flex položkou čistý textový box. Tento box je posléze obalen do anonymního blokového boxu (vlastnosti textového boxu jsou rovněž předány anonymnímu boxu, stejně jako v případě mřížkového rozložení) a stává se z něj flex položka. Vytvoření anonymní položky se nachází ve výpisu 7.13. Tato část se rovněž vyskytuje v metodě `createAnonymousBox()`.

```
if (parent.display == ElementBox.DISPLAY_FLEX) {
        anbox = new FlexItem(anelem, child.getVisualContext().create());
        anbox.setStyle(createAnonymousStyle("block"));
```

Výpis 7.13: Vytvoření instance anonymní flexibilní položky.

Dále je pak v příslušném konstruktoru flexibilního kontejneru vytvořena instance layout manageru, přesněji `FlexBoxLayoutManager`. Instance je poté uložena v atributu `typeOfLayout`.

Styly CSS pro boxy flexibilního rozložení

V případě flexibilního rozložení jsou dostupné styly příslušných boxů načítány obdobně jako v případě mřížkového rozložení. Tedy je využita metoda `setStyle()` k přiřazení nových stylů a ke zpracování konkrétního stylu je znova použita konstrukce `style.getProperty("zadany-styl")`. Rovněž je zde možné využít instanci třídy `CSSDecoder` a její metodu `getLength()` k převodu do pixelových jednotek.

Kromě toho i zde platí, že všechny načtené styly jsou uloženy v příslušných atributech tříd a nespecifikovaným stylům jsou automaticky přidělovány jejich výchozí hodnoty podle oficiální specifikace CSS pro flexibilní rozložení.

7.2.2 Detekce hlavní a vedlejší osy

Prvním krokem je nastavení počátečního hlavního a vedlejšího rozměru kontejneru. To se provádí na základě zadané šířky/výšky či na základě obsahu kontejneru (zde již je znám nějaký minimální obsah kontejneru). Tyto rozměry jsou dále v případě potřeby upravovány, nejedná se totiž o finální velikost.

Následuje vytvoření struktury `ArrayListu`, do kterého jsou uloženy všechny položky kontejneru. Všem položkám jsou rovnou zakázané vlastnosti `clear` a `float`, protože dle oficiální specifikace na ně nemají mít žádný vliv.

Nyní dochází k detekci směru kontejneru na základě vlastnosti `flex-direction`. Tato detekce je znázorněna na výpisu 7.14. Jakmile se vyhodnotí, o jaký kontejner jde, je zavolána metoda buď pro horizontální směr, nebo vertikální směr. Tyto metody mají jako svůj parametr *ArrayList* se všemi položkami kontejneru. Metody jsou rozděleny podle směru z důvodu toho, že části nejsou úplně stejné a v určitých segmentech jsou rozdílné.

```
public boolean isRowContainer() {
        if (flexDirection == FLEX_DIRECTION_ROW || flexDirection ==
            FLEX_DIRECTION_ROW_REVERSE)
                return true;
        else
                return false;
}
```

Výpis 7.14: Detekce horizontálního flex kontejneru. V případě vertikálního kontejneru je postup obdobný.

Nicméně směr kontejneru dle specifikace ještě může ovlivnit vlastnost `writing-mode`[2], která mění směr textu uvnitř kontejneru, ale tato vlastnost není vůbec zahrnuta v knihovně `jStyleParser`.

Pořadí flexibilních položek na základě vlastnosti `order`

Před samotným nastavením rozměrů flex položek v kapitole 7.2.3 je nutné ještě provést prohození položek na základě vlastnosti `order`. Jelikož jsou jednotlivé položky stále uloženy v `ArrayListu`, lze na tento list jednoduše aplikovat statickou metodu pro řazení `sort()`, která se nachází ve třídě `Collections`. Nicméně je ještě důležité ve třídě pro flex položky provést implementaci rozhraní `Comparable` a upravit metodu `compareTo()`, aby nedocházelo k porovnávání instancí objektů, ale naopak se mezi sebou porovnávaly hodnoty vlastnosti `order`. Upravená metoda `compareTo()` je k vidění ve výpisu 7.15. Položky jsou následně řazeny vzestupně, tedy položka s nejmenší hodnotou vlastnosti `order` bude v listu jako první.

[2] https://developer.mozilla.org/en-US/docs/Web/CSS/writing-mode

```
@Override
public int compareTo(FlexItem i) {
        return this.flexOrderValue - i.getFlexOrderValue();
}
```

Výpis 7.15: Přepsání metody `compareTo()` k porovnávání na základě vlastnosti `order`.

Bohužel kvůli nedostatku knihovny `jStyleParser` nelze pracovat se zápornými hodnotami vlastnosti `order`. V případě zadané záporné hodnoty je knihovnou vrácena hodnota *null*, pokud je tato hodnota detekována, je nastaven order položky na nula. Jakmile se přidá do knihovny `jStyleParser` práce se zápornými hodnotami pro tuto vlastnost, lze již snadno doimplementovat správné chování záporných hodnot.

7.2.3 Nastavení rozměrů flex položek

Jak už bylo dříve zmíněno, hlavní rozměr (velikost) položek je důležitý při pozicování, vykreslování a k nastavení hlavního rozměru vertikálního kontejneru. Tento rozměr zastupuje atribut `hypotheticalMainSize` ve třídě `FlexItem`. K získání tohoto rozměru je zapotřebí několika postupných kroků.

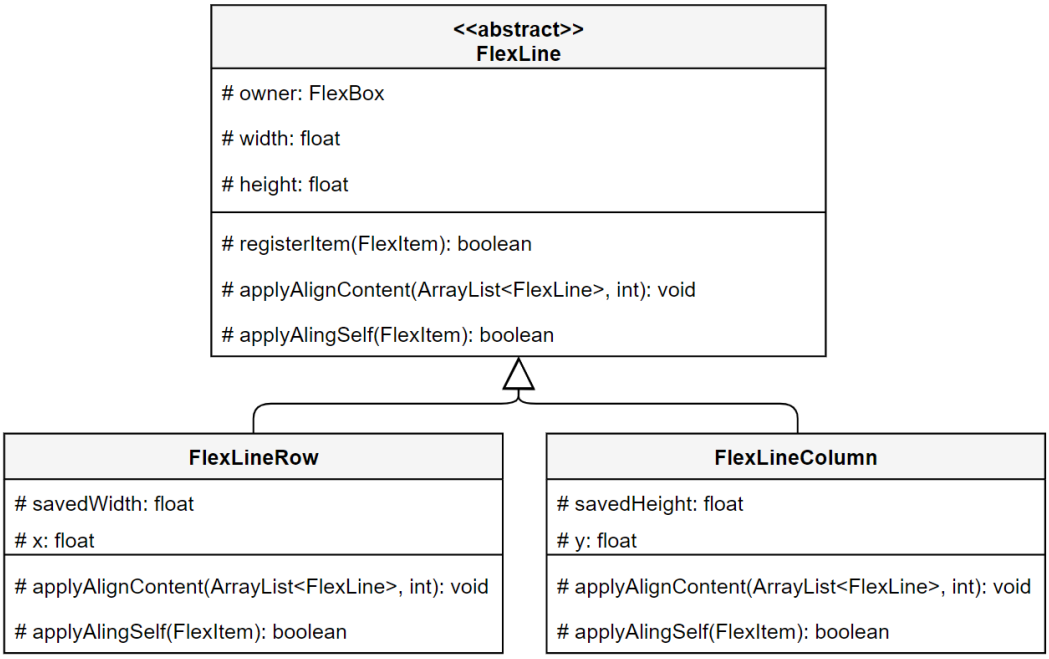
Prvním krokem je zpracování vlastnosti `flex-basis`, tu lze zadat několika variantami. Pokud by byla zadána v číselné nebo procentuální podobě, je zpracována a uložena do příslušného atributu. Dále se provádí kontrola vůči vlastnosti `width` respektive `height` a to tak, že pokud byla šířka či výška větší než hodnota z vlastnosti `flex-basis` je rozměr poupraven na hodnotu šířky či výšky v závislosti směru hlavní osy. Pokud by byla zadána jako hodnota *auto* nebo nezadána vůbec, opět jsou využity vlastnosti šířky či výšky, kdy se vezmou jako hlavní rozměr. Nicméně ani šířka nebo výška nemusí být zadána, pokud by tento stav nastal, tak je jako hlavní rozměr brán svůj obsah. Dalším krokem je pak ohraničení hlavního rozměru již zmíněnými minimálními a maximálními hodnotami tak, že v cyklu jsou postupně procházeny všechny položky. K tomuto procesu je využita třída `Dimension` z knihovny CSSBox s atributy `min_size` (představující minimální velikost boxu) a `max_size` (představující maximální velikost boxu). Pokud nejsou ohraničující hodnoty zadány, v atributech je ve výchozím nastavení nastavena hodnota -1 a k ohraničení nedochází. Další činnosti s hlavním rozměrem jsou prováděny v následujících krocích algoritmu.

Co se týče vedlejšího rozměru (velikosti), ten lze snadno získat z vlastností `width`, `height` a jejich variant na základě orientace hlavní osy.

7.2.4 Rozčlenění položek mezi flex linky

Před samotným rozdělováním jednotlivých položek mezi flex linky byla nejprve vytvořena struktura pro flex linky, jak naznačuje diagram tříd 7.1. Struktura zahrnuje abstraktní třídu `FlexLine` představující sklad pro položky patřící do stejné linky. Z této třídy dědí další dvě třídy `FlexLineRow` a `FlexLineColumn`, které představují flex linky v závislosti na směru kontejneru (horizontální či vertikální). V těchto třídách se dále podle algoritmu provádí i zarovnání položek v dané lince.

Samotné vytváření flex linek vykonává metoda `createAndFillLines()` z třídy `FlexBox`. Tato metoda v první řadě založí počáteční flex linku kontejneru a uloží ji do struktury *ArrayList* sloužící pro práci s flex linkami. Následně jsou procházeny položky z *ArrayListu* položek, kdy se na každou položku volá metoda `registerItem()` z abstraktní třídy `FlexLine`. Ta registruje položku a přiřadí ji do dané linky, dále ještě detekuje případné překročení zby-



Obrázek 7.1: Diagram tříd znázorňující implementaci flex linek v knihovně CSSBox, jsou zde vyobrazeny i některé atributy a metody těchto tříd.

lého prostoru linky. Pokud je zbylý prostor překročen, je vytvořena nová flex linka (instance třídy `FlexLineRow` nebo `FlexLineColumn`) a položka, která se již nevešla do předcházející linky, bude následně vložena do nové linky. Tento proces je nastíněn ve výpisu 7.16.

```
if (!line.registerItem(Item)) {
        FlexLine newLine;
        if (isRowContainer())
                newLine = new FlexLineRow(this);
        else
                newLine = new FlexLineColumn(this);
        lines.add(newLine);
        newLine.registerItem(Item);
}
```

Výpis 7.16: Vytvoření nové flex linky na základě směru kontejneru a vložení položky do ní, pokud se již položka nevejde do předcházející.

Linkám je poté nastavován i vedlejší rozměr. Jedná-li se o jednořádkový kontejner, flex lince je nastaven jako vedlejší rozměr vedlejší rozměr kontejneru. Jinak se nastavuje na základě největšího vedlejšího rozměru položky v dané flex lince, to je důležité pro další kroky, kdy se bude zpracovávat další zarovnání položek (například vlastnost `align-items`).

7.2.5 Aplikace flex faktorů

Flex faktory představují vlastnosti `flex-shrink` a `flex-grow`, ty byly již představeny v kapitole 4.3.3. Tyto faktory dokážou modifikovat hlavní rozměr flex položek, který je poté

ovlivněn oproti hlavnímu rozměru kontejneru. Aktuálně jsou jednotlivé položky umístěny v odpovídajících linkách, tudíž se budou aplikovat flex faktory postupně na všechny linky.

V případě horizontálního kontejneru je důležité zpracovat flex faktory před samotným vykreslením obsahu položek, protože je potřeba znát šířka položek, která aktuálně nemusí být finální, protože flex faktory ji stále mohou ovlivnit. Hlavní rozměr je zde uložen v atributu `hypotheticalMainSize` a je prováděna modifikace vůči tomuto atributu.

V případě vertikálního kontejneru je hlavní rozměr již uložen v atributu `content.height` a je prováděna modifikace vůči němu.

Vlastnost `flex-shrink`

Implementace vlastnosti `flex-shrink` je provedena tak, že jsou nejprve v cyklu postupně procházeny flex linky, ve kterých se provádí součet hodnot vlastnosti `flex-shrink` a hlavních rozměrů položek (zde jsou zahrnuty i vlastnosti `border, margin` a `padding`). Následně je spočítán zbylý prostor v dané lince pomocí vzorce 7.2.

$$remain\_size = \frac{main\_size\_of\_items\_in\_line - main\_size}{flex\_shrink\_sum} \tag{7.2}$$

Význam proměnných je zde popsán na jednom místě pro vzorce flex faktorů 7.2 a 7.3.

- *remain_size* – zbývající volný prostor
- *main_size_of_items_in_line* – suma hlavních rozměrů položek v dané flex lince
- *main_size* – hlavní rozměr kontejneru
- *flex_shrink_sum* – suma hodnot vlastnosti `flex-shrink` v dané flex lince
- *flex_grow_sum* – suma hodnot vlastnosti `flex-grow` v dané flex lince

Po získání zbylého prostoru (tedy proměnné `remain_size`) je provedeno násobení s hodnotou vlastnosti `flex-shrink` u každé položky. Tento mezivýsledek (pokud není záporný) je poté odečten od hlavního rozměru dané položky. V situaci, kdy by tento mezivýsledek vyšel záporně, by znamenalo, že jednotlivé položky v dané lince nepřesahují hlavní rozměr kontejneru. To je pokaždé kontrolováno podmínkou, pokud by tato podmínka vyšla kladně, může dojít k předčasnému ukončení zpracování vlastnosti `flex-shrink`.

Hodnoty jsou v posledním kroku následně ohraničeny minimálním rozměrem v souvislosti s hlavní osou, kdy je využit již zmíněný atribut `min_size` z třídy `Dimension`.

Vlastnost `flex-grow`

Implementace vlastnosti `flex-grow` je rovněž provedena za pomoci cyklu, kdy jsou postupně procházeny jednotlivé flex linky. Pro každou linku se také provádí součet hlavních rozměrů položek s rozdílem, že se zde ale provádí součet vlastnosti `flex-grow`. Následně je zase spočítám zbylý prostor v dané lince podle vzorce 7.3.

$$remain\_size = \frac{main\_size - main\_size\_of\_items\_in\_line}{flex\_grow\_sum} \tag{7.3}$$

Jakmile je spočítán zbylý prostor, je opět provedeno násobení, ale s hodnotou vlastnosti `flex-grow` u každé položky. Rovněž je získán mezivýsledek a ten, pokud není záporný, je přičten k hlavnímu rozměru dané položky. Když vyjde mezivýsledek záporný, představuje to, že položky uvnitř dané linky přesahují hlavní rozměr kontejneru, a to je opět kontrolováno podmínkou, která může vyvolat předčasné ukončení zpracování vlastnosti `flex-grow`.

Hodnoty jsou v posledním kroku následně ohraničeny maximálním rozměrem v souvislosti s hlavní osou, kdy je využit již zmíněný atribut `max_size` z třídy `Dimension`.

7.2.6 Zobrazení obsahu položek

K vykreslování položek (zobrazení obsahu) se také používá metoda `doLayout()` z abstraktní třídy `ElementBox`. Jak již bylo zmíněno, metoda obsahuje tři parametry, přičemž první parametr představuje, s jakou předanou šířkou dané položky má počítat. Tudíž v případě horizontálního kontejneru je metodě předána šířka položek v podobě atributu `hypotheticalMainSize`, naopak v případě vertikálního kontejneru je metodě předán atribut `content.width` (oba atributy představují šířku bez vlastností `border, margin` a `padding`). Díky tomuto způsobu se rovněž nemusí zasahovat do třídy `BlockBoxLayoutManager` v podobě upřesnění, zda se jedná či nejedná o flex položky.

Po dokončení zpracování v metodě `doLayout()` je ještě všem položkám upravován margin (vlastnost `margin-right`), protože je nastavován do pravého kraje nadřazeného boxu. Úprava spočívá v tom, že se znovu získá hodnota vlastnosti `margin-right` a ta je poté nastavena. V posledním kroku jsou nově získané hodnoty uloženy v atributech `bounds` a `content`, které jsou využity při pozicování.

7.2.7 Pozicování a zarovnání flex položek a linek

Tento krok algoritmu zpracovává vlastnosti pro pozicování a zarovnání. Jedná se o vlastnosti `align-content`, `align-items`, `align-self` a `justify-content`. Zpracování těchto vlastností je rozděleno do jednotlivých metod dle jejich funkce. Nejprve se určuje vedlejší rozměr daných linek na základě vlastnosti `align-content`. Další část se věnuje pozicím položek vůči vedlejší ose, tedy vlastnostem `align-items` a `align-self`. V poslední části je provedena implementace vlastnosti `justify-content`, která ještě dále může upravovat hlavní osu.

Implementace vlastnosti `align-content`

V první řadě je nutné flex linkám nastavit jejich pozici a rozměr v kontejneru na základě této vlastnosti. Implementace je provedena v několika krocích, kdy je nejprve nutné nastavit každé lince souřadnici uvnitř vedlejší osy (nastavení se provádí vůči atributům `x` a `y`, na základě typu flex linky, ty byly již naznačeny v diagramu tříd 7.1). Souřadnice dané flex linky je spočtena tak, že se sčítají vedlejší rozměry předchozích linek. Dále je pak dopočítáno zbylé místo na základě vedlejšího rozměru linek a kontejneru. Způsob spočítání zbylého místa je ukázán ve výpisu 7.17, tento princip je implementován ve třídě `FlexLineRow` a obdobně je proveden i ve třídě `FlexLineColumn`.

```
float remainingHeightSpace = owner.crossSize;
for (FlexLine flexLine : lines) {
        FlexLineRow line = (FlexLineRow) flexLine;
        if (line.savedHeight == -1)
                remainingHeightSpace -= line.getHeight();
        else
                remainingHeightSpace -= line.savedHeight;
}
```

Výpis 7.17: Způsob dopočítání zbylého místa, které pak dále koriguje v dalších výpočtech u vlastnosti `align-content`.

Aktuálně jsou známy všechny potřebné hodnoty pro vyhodnocování vlastnosti `align-content`. Chování jednotlivých hodnot je obecně implementováno podle specifikace. Dobré je zmínit, že v případě zadaných hodnot `space-between` nebo `space-around` a jednořádkového kontejneru je chování odlišné oproti víceřádkovému kontejneru. Tento případ je řešen a ukázán ve výpisu 7.18 pro hodnotu `space-around`. Pro hodnotu `space-between` je to řešeno obdobně.

```
if(owner.alignContent == FlexBox.ALIGN_CONTENT_SPACE_AROUND){
  if(countOfPreviousLines == 0)
    setY(getY() + remainingHeightSpace / (2 * lines.size()));
  else
    setY(getY() + remainingHeightSpace / (2 * lines.size()) +
        (countOfPreviousLines) * remainingHeightSpace / lines.size() + 1);
}
```

Výpis 7.18: Ukázka vyhodnocení vlastnosti `align-content`, kdy je zadána hodnota `space-around`.

V případě zadané hodnoty `stretch` je nutné si pomáhat pomocnou proměnnou pro ukládání vedlejšího rozměru, protože by jinak nedocházelo k rovnoměrné distribuci zbylého místa uvnitř kontejneru.

Implementace vlastností `align-items` a `align-self`

V této části se provádí implementace vlastností `align-items` a `align-self`, které ovlivňují zarovnávání položek vůči vedlejšímu rozměru jednotlivých flex linek. Při zpracování těchto vlastností je nutné brát v potaz i vlastnost `flex-direction`, která může měnit výsledný efekt v závislosti toho, jak je nastavena. Ukládání pozic jednotlivých položek prováděno za pomoci metody `setPosition(x, y)`. Nicméně hlavní rozměr položek (v tomto případě souřadnice `x`) v tomto kroku je nezajímavý, tudíž je aktuálně ukládán s nulovou hodnotou a bude zpracován v dalším kroku u vlastnosti `justify-content`.

První položce v lince se vždy jednoduše nastaví její vedlejší rozměr a poté se pro ni zjišťuje jaké vlastnosti má zadané, protože dle specifikace má vlastnost `align-self` přednost před vlastností `align-items`. Jakmile jsou dané vlastnosti zpracovány (zjednodušený příklad zpracování hodnot `flex-end` a `center` je ukázán ve výpisu 7.19), jde se na další položku v lince. Princip funguje podobně, nicméně jakákoliv další položka v lince může již ovlivnit vedlejší rozměry předchozích položek. Je proto nutné znovu projít předchozí položky v lince a donastavit jim správný rozměr.

```
if (owner.alignItems == FlexBox.ALIGN_ITEMS_FLEX_END) {
  for (FlexItem flexItem : itemsInLine) {
    flexItem.setPosition(flexItem.bounds.x, y + getHeight() -
        flexItem.getHeight());
  }
} else if (owner.alignItems == FlexBox.ALIGN_ITEMS_CENTER) {
  for (FlexItem flexItem : itemsInLine) {
    flexItem.setPosition(flexItem.bounds.x, y + (getHeight() -
        flexItem.getHeight()) / 2);
  }
}
```

Výpis 7.19: Ukázka vyhodnocení vlastnosti `align-items` pro hodnoty `flex-end` a `center`.

Za speciální hodnotu u těchto vlastností lze považovat hodnotu `baseline`, kde její princip zarovnání funguje na základě spodní hranice řádku. Díky jejímu chování může docházet i ke zvětšování vedlejšího rozměru linky, případně i kontejneru. V její implementaci byla využita i metoda `getBaselineOffset()` z třídy `VisualContext`, která získává maximální vzdálenost aktuálního písma v závislosti od horní pozice dané linky. Tuto hodnotu lze pouze aplikovat na horizontálním kontejneru, v případě vertikálního se projeví jako hodnota `flex-start`.

Implementace vlastnosti `justify-content`

Poslední částí tohoto kroku je implementace vlastnosti `justify-content`. Tato vlastnost určuje a ovlivňuje pozice položek vůči hlavnímu rozměru kontejneru.

Nejprve jsou dané metodě předávány jednotlivé linky s položkami, kdy je v první řadě spočítán celkový hlavní rozměr pro každou linku tak, že jsou postupně sčítány hlavní rozměry položek. Dále je potřeba kontrola v závislosti na vlastnosti `flex-direction`, jestli se náhodou nejedná o otočený kontejner (jen pro upřesnění hodnoty `row-reverse` a `column-reverse`), protože podle toho jsou pak položky v lince zpracovávány od konce nebo od začátku, kromě hodnoty `center`, u které to nemá žádný vliv. Nyní již lze nastavit správné souřadnice jednotlivým položkám podle vlastnosti `justify-content`. Ve výpisu 7.20 je ukázáno, jak se nastavuje pozice položek v závislosti hodnoty `center`.

```
if (owner.justifyContent == FlexBox.JUSTIFY_CONTENT_CENTER) {
        float halfOfRemainSpace = (owner.mainSize) / 2;
        item.setPosition(halfOfRemainSpace - (totalWidthOfItems / 2) +
            widthOfPreviousItems, item.bounds.y);
}
```

Výpis 7.20: Nastavení pozice položkám u vlastnosti `justify-content` a její hodnoty `center`.

V poslední řadě je nutné zmínit, že může nastat situace, kdy flex linka obsahuje pouze jednu položku. V této situaci samozřejmě lze normálně aplikovat vlastnost `justify-content` a některé její hodnoty se chovají jinak než by měly (ostatní vykazují běžné chování). V případě hodnoty `space-between` se tato hodnota chová jako `flex-start` a v případě hodnoty `space-around` se tato hodnota chová jako `center`.

7.2.8 Výsledná velikost kontejneru

Posledním krokem flexbox algoritmu je zajištění správného zobrazení flex kontejneru v knihovně CSSBox. Po vykreslení flex položek a zpracování případně zadaných zarovnávacích a pozicovacích vlastností je provedeno nastavení rozměrů kontejneru. Kontejneru jsou opět nastavovány jeho atriuty `bounds` a `content` (přesněji jejich šířky a výšky). V důsledku směru kontejneru je pak těmto atributům nastaven hlavní nebo vedlejší rozměr. Tyto rozměry jsou ještě upraveny o vlastnosti `border, margin` a `padding`.

7.3 Další provedené změny

V rámci práce byly provedeny i změny kódu kvůli přehlednosti a jejich významu vůči dané třídě. Tedy metody týkající se spíše výpočtu rozložení byly přesunuty do příslušných layout managerů apod. V případě třídy `BlockBox` by se tato změna ve výsledku promítla spíše negativně, protože je hojně využívána a muselo by se provést ještě více zásahů. U třídy `InlineBox` již nějaká změna proběhla a metody byly přesunuty do příslušného manageru.

Byla provedena i modifikace metody `alignLineHorizontally()` ve třídě `BlockBox`, kde byla do metody přidána jedna podmínka pro položky mřížky. A to z důvodu, že text uvnitř položek se nezarovnával vůči nim, ale byl zarovnán vůči oknu aplikace.

7.4 Odhalené nedostatky

Jedním z nedostatků je zpracování záporné hodnoty u vlastnosti `order` v knihovně `jStyleParser`, knihovna nedokáže zpracovat zápornou hodnotu, a tím pádem když je zadána, vrací se hodnota `null`. V některých případech nefunguje správně ani záporná hodnota u vlastnosti `z-index`. Nulová podpora je u vlastnosti `writing-mode`, která má poměrně velký vliv na flexibilní kontejnery. To samé platí i pro vlastnost `direction`[3], která dokáže ovlivňovat směr textu (tato vlastnost se dost objevovala i v testovací sadě a byla vypínána).

Odhalené nedostatky budou nahlášeny v podobě reportů (*Issues*) na oficiální *github* stránky knihoven `CSSBox`[4] a `jStyleParser`[5].

Co se týče mřížkového a flexibilního rozložení, nepodařilo se implementovat jejich vzájemnou integraci, což znamená, že například nelze zobrazit flexibilní kontejner uvnitř mřížkového kontejneru. Dále se nepodařilo implementovat všechny různé kombinace hodnot při vytváření mřížky, jelikož jich je poměrně dost. V některých případech byly nalezeny i problémy s obrázky, které vykazují jiné chování, než by měly. Pro některé případy se problém podařil vyřešit, nicméně v několika případech stále vykazují odlišné chování.

7.5 Návrhy na další rozšíření knihovny CSSBox

Takovým obecným návrhem pro další možnosti rozšíření funkčnosti knihovny CSSBox je určitě se zabývat nahlášenými odhalenými nedostatky, protože skoro pokaždé se najde něco, co již nefunguje, funguje jinak než by mělo nebo třeba již nevyhovuje oficiální specifikaci, která se mění či vyvíjí. V případě mřížkového a flexibilního rozložení je možné dále knihovnu CSSBox rozšířit o jejich inline kontejnery a celkově se zaměřit na oblast inline boxů. Jelikož

[3] https://developer.mozilla.org/en-US/docs/Web/CSS/direction
[4] https://github.com/radkovo/CSSBox/issues
[5] https://github.com/radkovo/jStyleParser/issues

je mřížkové rozložení poměrně novým typem pozicování, přináší s sebou i pokročilé techniky pozicování obsahu, kterých je mnoho a není jednoduché je vychytat. Bylo by dobré se také zaměřit na větší provázanost těchto rozložení mezi sebou a blokovým či inline rozložením. Velkým přínosem by také byla podpora JavaScriptu při testování, díky které by mohlo být použito mnohem více testů k ověření funkcionality.

Kapitola 8

Testování

Tato kapitola je věnována testování mřížkového a flexibilního rozložení. V začátku kapitoly je popsána testovací sada od konsorcia W3C, dále jsou přiblíženy dva způsoby testování, které s touto testovací sadou pracují. Je zde i krátce nastíněna práce s vlastními testovacími příklady. Závěr kapitoly je poté věnován výsledkům a zhodnocením provedeného testování.

8.1 Testovací sada

Pro otestování implementace algoritmů Grid layout a Flexbox byla využita oficiální sada od konsorcia W3C. Tato sada je velmi rozsáhlá a obsahuje plno testů pro různé úrovně modulů jazyka CSS. Sadu lze najít na oficiálních stránkách konsorcia[1]. Z této velké množiny byla vybrána podmnožina, která se zabývá mřížkovým[2] a flexibilním[3] rozložením. Mezi testy se nachází větší množství testů, které jsou vyhodnocovány na základě JavaScriptu, bohužel ten není podporován knihovnou CSSBox, proto tyto testy byly vynechány z testovací sady, protože při spuštění takového testu se výsledek jeví jako validní a po vizuální stránce je zobrazena pouze prázdná stránka. V testech se objevují i vlastnosti, které knihovna CSSBox nepodporuje, ale tyto testy byly ponechány, aby bylo možné zjistit, o jaké vlastnosti jde, ty poté budou nahlášeny.

První způsob využití testovací sady

Při prvním využití testovací sady bylo pracováno s testovacím rozhraním knihovny CSSBox. Přesněji se jedná o třídu `ReferenceComparisonTest`, která spouští automatické testování. Pro tento typ testování byly využity dvě starší testovací sady, které toto rozhraní akceptuje, tedy sada[4] pro mřížkové rozložení a sada[5] pro flexibilní rozložení. Obě testovací sady obsahují soubor *reftest-toc.htm*, který obsahuje odkazy na všechny testy v dané sadě. V souboru *test_reference.csv* jsou tyto testy ještě nastaveny s jejich přípustnými odchylkami, kdy odchylka může být v intervalu od nuly do jedné. Čím více se zvolená hodnota bude blížit k číslu jedna, tím je větší šance na projití testu. Poté již lze spustit automatické testování. Tento způsob testování je poměrně rychlý, ale stávalo se, že testy co neměly být akcepto-

[1] http://test.csswg.org/harness/
[2] http://test.csswg.org/suites/css-grid-1_dev/nightly-unstable/
[3] http://test.csswg.org/suites/css-flexbox-1_dev/nightly-unstable/
[4] https://github.com/jgraham/css-test-built/tree/master/css-grid-1_dev
[5] https://github.com/jgraham/css-test-built/tree/master/css-flexbox-1_dev/html

vány, tak akceptovány byly. Závěrem se tento způsob moc nevyužil a přešlo se na druhý způsob testování, který je sice pomalejší, ale více se zaměřuje vizuální stránku.

Druhý způsob využití testovací sady

Další typ testování se věnuje vizuální stránce daného testu. K tomuto procesu testování byl využit webový prohlížeč *Google Chrome* ve verzi 90.0 s nástrojem *DevTools*. Samotné testování probíhalo tak, že ve webovém prohlížeči byl spuštěn daný test s nástrojem *DevTools*, aby bylo možné vidět zdrojový kód testu a hlavně jednotlivé vlastnosti příslušných elementů. Dále byla spuštěna knihovna `CSSBox` pomocí demo aplikace, kterou zastupuje třída `BoxBrowser`, ta funguje na podobném principu jako webový prohlížeč a umožňuje si rovněž zobrazit jednotlivé vlastnosti daných elementů. Jakmile se v obou prohlížečích promítly testy, došlo k porovnávání vlastností mezi sebou, a také k rozhodnutí, zda daný test prošel nebo neprošel. Pokud daný test neprošel, došlo poté k přezkoumání, jaká byla příčina jeho neúspěchu.

Pro flexibilní rozložení bylo použito 667 testovacích příkladů z testovací sady. Toto číslo již zahrnuje všechny použité testy po vynechání testů, které se vyhodnocovaly na základě JavaScriptu. V případě mřížkového rozložení bylo použito 313 testovacích příkladů, rovněž zde platí, že toto číslo zastupuje testy bez použití JavaScriptu.

Tento způsob testování sice nebyl tak efektivní v rychlosti, ale na druhou stranu bylo možné vidět daný test s jednotlivými vlastnostmi daných elementů a případné chyby řešit.

8.2 Vlastní testovací příklady

V průběhu implementace byly využívány i vlastní testovací příklady, které dopomáhaly k odhalování chyb. Ty se především zaměřují na základní funkčnosti a jejich postupné modifikace. Díky nim byl například odhalen nedostatek knihovny `jStyleParser`, která neobsahuje podporu záporné hodnoty u vlastnosti `order`. Všechny vytvořené testovací příklady (kterých je šest a některé jsou tvořeny z několika podpříkladů) jsou součástí přiloženého CD disku a jsou k nalezení v adresáři *tests*, kdy jsou ještě rozděleny podle mřížkového a flexibilního rozložení.

8.3 Zhodnocení testování

Testy z druhého způsobu využití testovací sady byly postupně zaznamenávány v příslušných tabulkách programu *Microsoft Excel*. Tato tabulka je rovněž k nalezení v přiloženém CD ve složce *tests*. V tabulce je zapsán daný test, jeho výsledek a případná poznámka k němu (co by mohlo být neúspěchem testu nebo zda byla provedena změna nějaké části testu). Je zde veden i záznam představující celkový poměr úspěšných a neúspěšných testů jednotlivých rozložení.

Testování flexibilního rozložení dopomohlo k opravení několika chyb a otestování funkčnosti metody `doLayout()`. Statistiky u flexibilního rozložení si je možné prohlédnout v tabulce 8.1.

Flexibilní rozložení	
Počet testů	667
Počet úspěšných testů	508
Úspěšnost	76,16%

Tabulka 8.1: Výsledky testování pro flexibilní rozložení.

Z celkových 667 testů bylo úspěšných 508, což činí úspěšnost 76,16%. V těchto testech jsou zahrnuty i nepodporující vlastnosti apod. V případě modifikace testů (kdy byla například daná vlastnost vypnuta) se úspěšnost vyšplhala na 92,8%. Nicméně je důležité toto číslo brát s rezervou, protože na testu proběhla nějaká modifikace a reálně by se mělo počítat se 76% úspěšností.

Testování mřížkového rozložení rovněž testovalo funkčnost metody `doLayout()` a odhalilo několik nežádoucích případů při pracování s automatickými položkami. Statistiky u mřížkového rozložení jsou zobrazeny v tabulce 8.2.

Mřížkové rozložení	
Počet testů	313
Počet úspěšných testů	212
Úspěšnost	67,73%

Tabulka 8.2: Výsledky testování pro flexibilní rozložení.

Z celkových 313 testů bylo úspěšných 212, což ve výsledku vychází jako 67,73% úspěšnost. Rovněž v těchto testech jsou zahrnuty i nepodporující vlastnosti apod. V případě modifikace některých testů se úspěšnost ocitla na 70,29%. Rovněž je důležité brát toto číslo s rezervou a vycházet z původní úspěšnosti.

Díky testování byly odhaleny i některé nedostatky zmíněné v kapitole 7.4. Ve výsledku flexibilní rozložení vykazuje lepší výsledky, protože je trochu staršího původu a není tak rozmanité jako mřížkové rozložení, ve kterém se vyskytuje velká škála různých vlastností.

Kapitola 9

Závěr

Cílem této práce bylo prozkoumat architekturu zobrazovacího stroje CSSBox, prostudovat nové způsoby rozložení obsahu webových stránek pomocí jazyka CSS a navrhnout způsob integrace těchto způsobů rozložení obsahu do knihovny CSSBox. Nejprve byl krátce prostudován jazyk CSS, po kterém následovala podrobná studie mřížkového a flexibilního rozložení. Dále byla prozkoumána knihovna CSSBox a její aktuální struktura. Nicméně tato struktura se ukázala jako nesprávná, jelikož nebyla možná integrace nových rozložení a bylo kvůli tomu nutné modifikovat závislosti původního návrhu. Jedním z kroků bylo tedy navrhnout novou strukturu, která již umožňuje integraci nových rozložení do knihovny, a tu následně implementovat. Další krok se zabýval implementací mřížkového a flexibilního rozložení, kdy tyto rozložení byly poté otestovány na testovací sadě a vlastních testovacích příkladech. Na základě testování byly následně zhodnoceny dosažené výsledky. Nyní lze mřížkové a flexibilní rozložení zakomponovat do struktury knihovny CSSBox a experimentálně využívat, protože se již nemusejí vyskytovat v rozdělených větvích uvnitř repositáře na serveru *github*.

Literatura

[1] Andrew, R. *Get Ready for CSS Grid Layout*. 1st Edition. A Book Apart, 2016. ISBN 978-1-9375572-7-0.

[2] Atanassov, R., Brufau, O., Etemad, E. a Jr., T. A. *CSS Grid Layout Module Level 1*. 2020. [Online; navštíveno 30. 10. 2020]. Dostupné z: https://www.w3.org/TR/css-grid-1/.

[3] Baron, D., Etemad, E., Jr., T. A. a Atanassov, R. *CSS Flexible Box Layout Module Level 1*. 2018. [Online; navštíveno 9. 12. 2020]. Dostupné z: https://www.w3.org/TR/css-flexbox-1/.

[4] Burget, R. *CSSBox Manual*. 2020. [Online; navštíveno 25. 10. 2020]. Dostupné z: http://cssbox.sourceforge.net/manual/.

[5] Burget, R. *JStyleParser Manual*. 2020. [Online; navštíveno 27. 10. 2020]. Dostupné z: http://cssbox.sourceforge.net/jstyleparser/manual.php.

[6] Bříza, P. *Tvorba layoutu webu – teoretický úvod*. 2004. [Online; navštíveno 14. 12. 2020]. Dostupné z: https://www.interval.cz/clanky/tvorba-layoutu-webu-teoreticky-uvod/.

[7] Coyier, C. *A Complete Guide to Flexbox*. 2020. [Online; navštíveno 10. 11. 2020]. Dostupné z: https://css-tricks.com/snippets/css/a-guide-to-flexbox/.

[8] Coyier, C. *A Complete Guide to Grid*. 2020. [Online; navštíveno 6. 11. 2020]. Dostupné z: https://css-tricks.com/snippets/css/complete-guide-grid/.

[9] Etemad, E. *CSS Box Model Module Level 3*. 2020. [Online; navštíveno 13. 11. 2020]. Dostupné z: https://www.w3.org/TR/css-box-3/.

[10] Etemad, E., Rivoal, F. a Jr., T. A. *CSS Snapshot 2020*. 2020. [Online; navštíveno 11. 12. 2020]. Dostupné z: https://www.w3.org/TR/css-2020/.

[11] Meyer, A. E. a Weyl, E. *CSS: The Definitive Guide*. 4th Edition. O'Reilly Media, Inc., 2017. ISBN 978-1-449-39319-9.

[12] Michálek, M. *Vzhůru do CSS3*. Michálek Martin – Vzhůru dolů, [e-kniha], 2015. Dostupné z: https://www.vzhurudolu.cz/ebook-css3/.

[13] NOTHREM. *Grid, aneb revoluce tabulkového layoutu*. 2017. [Online; navštíveno 19. 11. 2020]. Dostupné z: http://css.chobits.ch/grid/.

[14] Novák, O. *Implementace mřížkového rozložení ve zobrazovacím stroji CSS*. Brno, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/21816/.

[15] Ondrák, L. *Implementace flexibilního rozložení ve zobrazovacím stroji CSS*. Brno, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/21425/.

[16] Pettit, N. *Which Layout? Static, Liquid, Adaptive, or Responsive*. 2014. [Online; navštíveno 14. 12. 2020]. Dostupné z: https://blog.teamtreehouse.com/which-page-layout.

[17] Terberová, H. *Co je to CSS - kaskádové styly*. 2020. [Online; navštíveno 10. 12. 2020]. Dostupné z: https://artster.cz/co-je-to-css/.

[18] W3C. *Cascading Style Sheets*. 2020. [Online; navštíveno 11. 12. 2020]. Dostupné z: https://www.w3.org/Style/CSS/.

[19] W3Schools. *CSS Tutorial*. 2020. [Online; navštíveno 9. 12. 2020]. Dostupné z: https://www.w3schools.com/css/css_syntax.asp.

Příloha A

Obsah CD

V přiloženém CD jsou k nalezení následující adresáře a soubory:

- *app* – spustitelný soubor aplikace ve formátu *jar*
- *cssbox* – rozšířené zdrojové kódy knihovny CSSBox
- *diff* – diff soubor s vytvořeným rozšířením
- *documentation* - technická zpráva se zdrojovými kódy
- *tests* – adresář s testy
  - *Flexible-box-layout* – vlastní testovací příklady pro flexibilní rozložení
  - *Grid-layout* – vlastní testovací příklady pro mřížkové rozložení
  - *tests.xlsx* – excel soubor s výsledky testů testovací sady
- *readme* – soubor readme s informacemi
- *xnovak2b-cssbox.pdf* – technická zpráva práce ve formátu *pdf*

Příloha B

Zprovoznění projektu

Nejprve je nutné stáhnout zdrojové kódy knihovny CSSBox. Ty lze najít na přiloženém CD nebo případně na serveru *github*. V případě *github* serveru stačí zadat příkaz:

```
$ git clone https://github.com/Lardalwen/CSSBox.git
```

Dále jsou potřeba zdrojové kódy knihovny jStyleParser. Tu lze rovněž stáhnout ze serveru *github* následujícím příkazem:

```
$ git clone https://github.com/Lardalwen/jStyleParser.git
```

Aktuálně jsou již připraveny všechny zdrojové kódy a následující příkazy budou pracovat s nástrojem *Maven*[1].

Nyní je nutné provést instalaci knihovny jStyleParser, ta se provede zadáním následujících příkazů:

```
$ cd jStyleParser
$ mvn install
```

Knihovna jStyleParser je úspěšně nainstalována a je tedy možné přejít k instalaci knihovny CSSBox, ta se nainstaluje pomocí následujících příkazů:

```
$ cd ../CSSBox
$ git checkout develop
$ mvn install
```

Po instalaci knihovny CSSBox je možné přistoupit k jejímu spuštění pomocí příkazu:

```
$ mvn exec:java -Dexec.mainClass="org.fit.cssbox.demo.BoxBrowser"
```

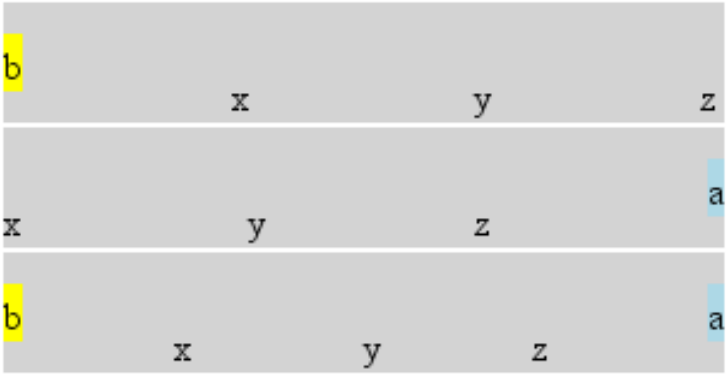
[1] https://maven.apache.org/index.html

Příloha C

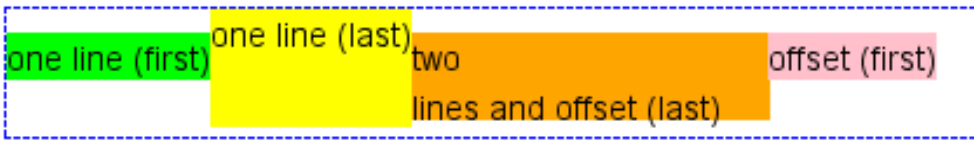
Příklady z testovací sady

Zde se nachází výsledky několika vybraných testů z testovací sady pro flexibilní a mřížkové rozložení, které jsou zobrazeny knihovnou CSSBox.

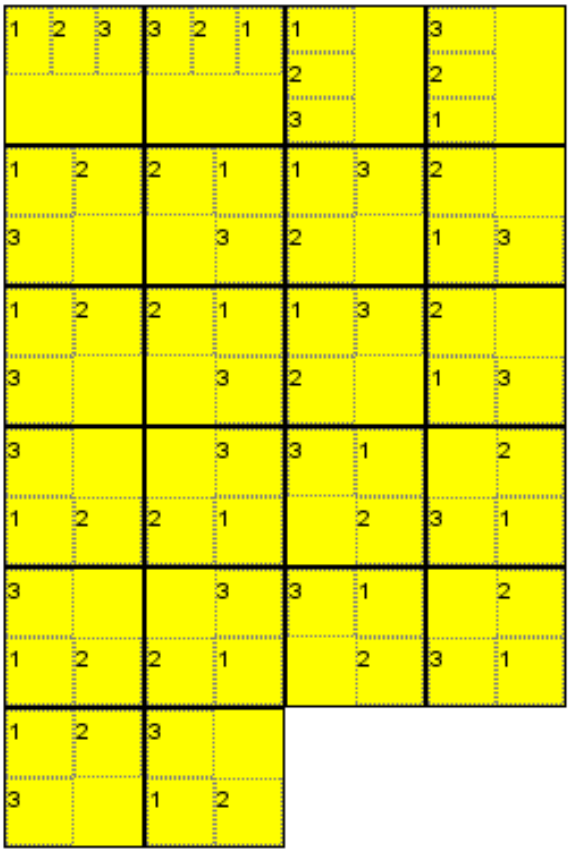
Flexibilní rozložení



Obrázek C.1: Test s vlastnostmi `justify-content` a `align-items`.

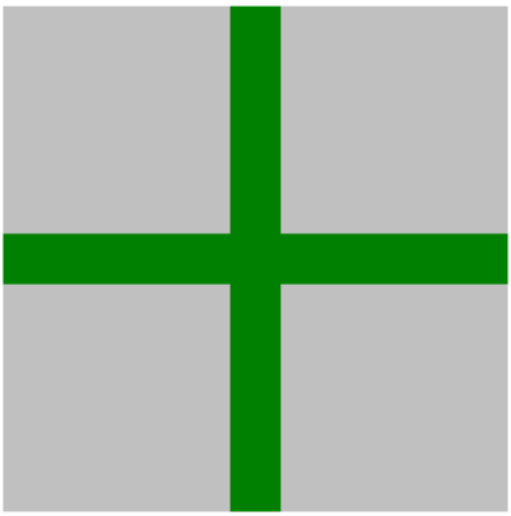


Obrázek C.2: Test na vlastnost `align-self`.

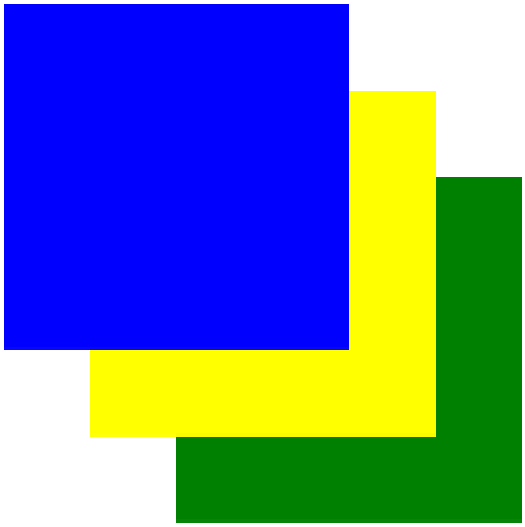


Obrázek C.3: Test na vlastnost `flex-flow`.

Mřížkové rozložení



Obrázek C.4: Test na vlastnost `grid-gap`.

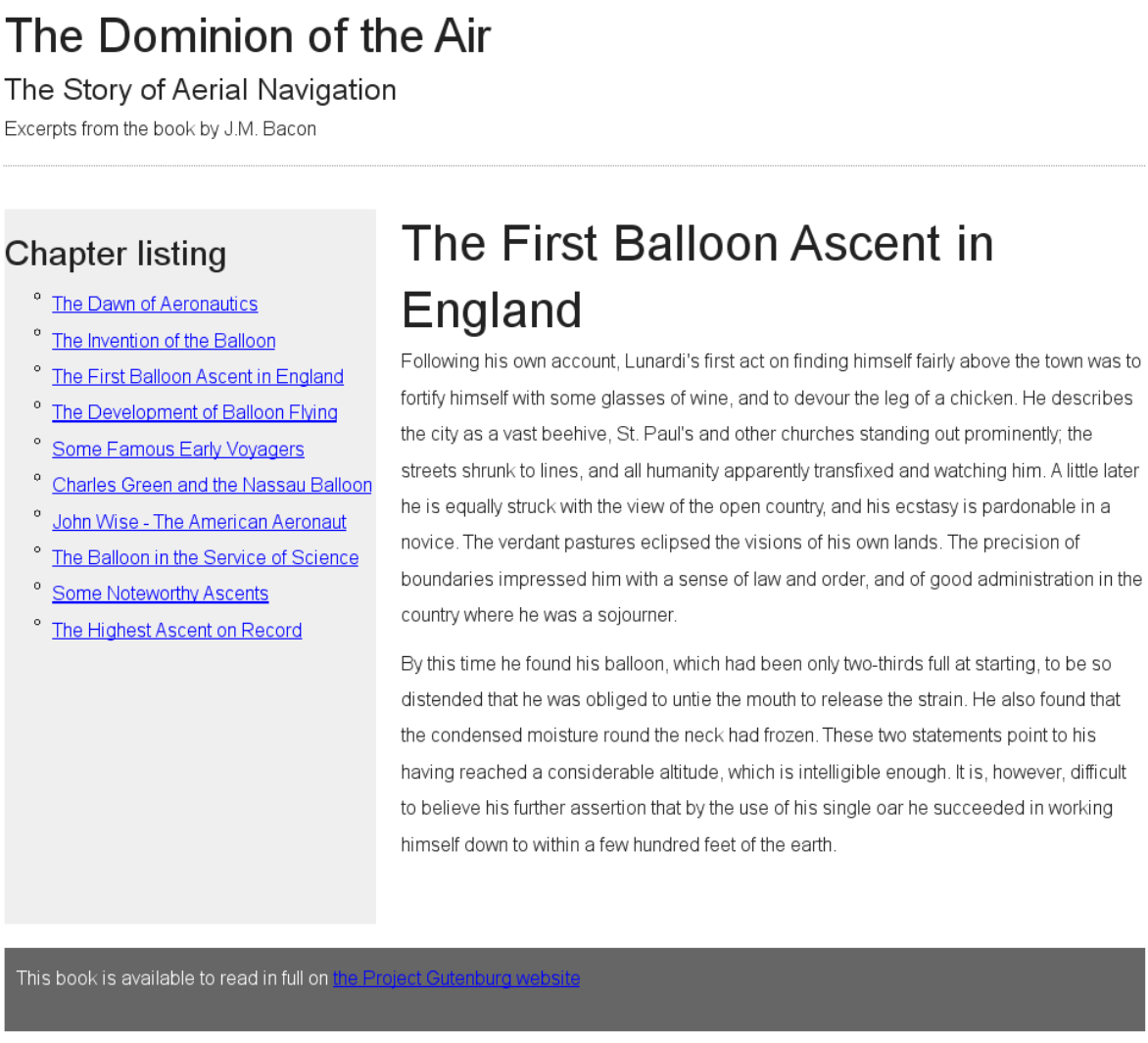


Obrázek C.5: Test na umístění položek do mřížky za použití vlastností `grid-row` a `grid-column`.

Příloha D

Vlastní testovací příklady

Zde jsou na ukázku přiloženy některé vlastní testovací příklady, které knihovna CSSBox zobrazuje. Ostatní příklady (včetně těchto) jsou k nalezení v přiloženém CD.



Obrázek D.1: Příklad webové stránky, která je vytvořena pomocí mřížkového rozložení.

Seamlessly visualize quality

Collaboratively administrate empowered markets via plug-and-play networks.

Completely Synergize

Dramatically engage seamlessly visualize quality intellectual capital without superior collaboration and idea-sharing.

Best in class

Imagine jumping into that boat, and just letting it sail wherever the wind takes you...

Obrázek D.2: Příklad webové stránky, která je vytvořena pomocí flexibilního rozložení.